BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PARAMETRIC CONTRACTS FOR CONCURRENCY IN JAVA PROGRAMS

INSTRUMENTACE JAVA PROGRAMŮ, KONTRAKTY PRO PARALELISMUS

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JAN ŽÁRSKÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. ALEŠ SMRČKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Žárský Jan, Bc.**
Program: Informační technologie a umělá inteligence Specializace: Bezpečnost informačních technologií
Název: **Instrumentace Java programů, kontrakty pro paralelismus**
Parametric Contracts for Concurrency in Java Programs
Kategorie: Analýza a testování softwaru

Zadání:

1. Nastudujte testování a dynamickou analýzu programů v jazyce Java. Nastudujte instrumentační framework RoadRunner pro dynamickou analýzu programů v jazyce Java. Seznamte se s kontrakty pro paralelismus.
2. Navrhněte nástroj pro jednoduchou instrumentaci testovaných programů. Navrhněte dynamický analyzátor pro sledování parametrických kontraktů.
3. Implementujte analyzátor v rámci RoadRunner.
4. Vytvořte testovací případy pro ověření hlavní funkcionality.

Literatura:

- DIAS Ricardo J., FERREIRA Carla, FIEDOR Jan, LOURENCO Joao, SMRČKA Aleš, SOUSA Diogo J. a VOJNAR Tomáš. Verifying Concurrent Programs Using Contracts. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). doi: 10.1109/ICST.2017.25
- Repozitář projektu RoadRunner Extended, https://pajda.fit.vutbr.cz/jct/roadrunnerX

Při obhajobě semestrální části projektu je požadováno:

- Studium a návrh analyzátoru

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 11. listopadu 2020

Abstract

Contracts for concurrency describe required atomicity of method sequences in concurrent programs. This work proposes a dynamic analyzer to verify programs written in Java against contracts for concurrency. The analyzer was designed to detect violations of parametric contracts with spoilers. The proposed analyzer was implemented as an extension to the RoadRunner framework. Support for accessing the method arguments and return values was added to RoadRunner as a part of the solution. The analyzer was fully implemented and verified on a set of testing programs.

Abstrakt

Kontrakty pro paralelismus slouží k vyjádření potřebné atomicity sekvencí metod ve vícevláknových programech. Tato práce se zaměřuje na implementaci dynamického analyzátoru, který verifikuje programy napsané v jazyce Java vůči kontraktům. Podporovány jsou parametrické kontrakty se spojlery. Analyzátor je implementován jako rozšíření frameworku RoadRunner. V rámci implementace analyzátoru byla do frameworku RoadRunner přidána podpora pro získávání argumentů metod a jejich návratových hodnot. Analyzátor byl plně implementován a jeho funkčnost byla ověřena na sadě testovacích programů.

Keywords

software verification, dynamic analysis, Java, contracts for concurrency, RoadRunner, instrumentation, Java bytecode, concurrent programming

Klíčová slova

verifikace softwaru, dynamická analýza, Java, kontrakty pro paralelismus, RoadRunner, instrumentace, Java bajtkód, vícevláknové programování

Reference

ŽÁRSKÝ, Jan. *Parametric Contracts for Concurrency in Java Programs.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D.

Rozšířený abstrakt

Při vývoji softwaru se běžně využívají knihovny nebo moduly vyvinuté jinými vývojáři. Při jejich integraci je zapotřebí dodržet pravidla stanovená autorem knihovny. Pravidla zahrnují syntaxi a sémantiku operací poskytovaných knihovnou. Ve vícevláknovém prostředí je ale zapotřebí dodržet dodatečné požadavky na synchronizaci vláken, která provádí operace poskytované danou knihovnou.

Kontrakty pro paralelismus slouží ke specifikaci omezení pro práci s knihovnou ve vícevláknových programech. Kontrakty specifikují, které sekvence operací musí být vykonávány atomicky, tedy bez toho, aby jiné vlákno provádělo souběžně jinou operaci. Existují dvě rozšíření, která upřesňují, za jakých podmínek je nutné dodržet atomicitu operací. Parametrické kontrakty reflektují datový tok mezi operacemi. Umožňují tak například vyjádřit, že dvě operace musí být prováděny atomicky pouze tehdy, pokud modifikují stejná data. Kontrakty se spojlery dovolují některým operacím probíhat souběžně, například pokud operace provádí pouze čtení sdílených dat. Kontrakty pro paralelismus lze sledovat za běhu programu a existuje metoda pro kontrolu jejich dodržování.

Cílem této práce je vytvořit dynamický analyzátor, který sleduje dodržování parametrických kontraktů se spojlery. Analyzátor pracuje s vícevláknovými programy v jazyce Java. Využívá frameworku RoadRunner, který provádí instrumentaci programů pro zkoumání chování programů za běhu. RoadRunner vkládá instrukce do bajtkódu programu, které pak za běhu zasílají analyzátoru události o volaných metodách, přístupech do paměti, synchronizaci vláken a podobně.

Vstupem analyzátoru je konfigurační soubor s definicí kontraktu, který určuje sekvence, které budou detekované analyzátorem. Sledovaný program je následně instrumentován frameworkem RoadRunner. Instrumentace volání metod byla v rámci práce rozšířena o získávání argumentů metod a jejich návratových hodnot. Instrumentovaný program je následně spuštěn. Analyzátor pro sledování kontraktů pro paralelismus konzumuje události spojené s voláním metod a synchronizací vláken. Na základě těchto událostí jsou detekovány sekvence metod a případná porušení kontraktu. Analyzátor si pro každé vlákno programu udržuje naposledy detekované sekvence metod. Pro každé vlákno a zámek si také udržuje vektorové hodiny nesoucí informace o vzájemné synchronizaci vláken. Jakmile je detekována celá sekvence, analyzátor na základě nedávných sekvencí v jiných vláknech a vektorových hodinách vyhodnotí, zda nedošlo k proložení sekvencí tak, aby byl porušen kontrakt. Díky využití vektorových hodin dokáže analyzátor odhalit proložení metod, ke kterému nedošlo přímo v daném běhu, ale může k němu dojít v podobných bězích.

Při návrhu analyzátoru byly zohledněny výsledky existujících prototypových implementací a schopnosti analyzátoru byly záměrně omezeny. Analyzátor tak klade dodatečné požadavky jak na kontrakty, tak na programy, které dokáže sledovat. Kontrakty musí splňovat následující podmínku: hodnoty všech parametrů kontraktu jsou určeny voláním první metody kontraktu. Tato podmínka zabraňuje zbytečné duplikaci sledovaných sekvencí. Analyzované programy nesmí obsahovat zanořená volání metod sledovaných v rámci kontraktu.

Při implementaci byly využity principy funkcionálního programování, zejména neměnné (immutable) objekty postavené na knihovně Vavr nebo funkce vyššího řádu. Jednotlivé části analyzátoru byly otestovány pomocí jednotkových testů, analyzátor jako celek pomocí Bash skriptů.

Výsledkem práce je plně funkční analyzátor parametrických kontraktů se spojlery. Změny v instrumentaci mohou být využity dalšími analyzátory vyžadujícími argumenty metod a návratové hodnoty. Jednotlivé části analyzátoru mohou být v budoucnu optimalizovány

s ohledem na rychlost. Funkcionální implementace analyzátoru umožňuje snadnou paralelizaci kontroly kontraktů. Dalšího zlepšení výkonu lze dosáhnout lepší definicí podmínek, za kterých lze zahazovat detekované sekvence metod. Analyzátor lze rozšířit o vkládání šumu pro detekci méně obvyklých chyb.

Parametric Contracts for Concurrency in Java Programs

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Aleš Smrčka, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this project.

.....
Jan Žárský
May 17, 2021

Acknowledgements

I would like to thank Ing. Aleš Smrčka, Ph.D., for valuable advice that helped me with the implementation and writing of this thesis.

Contents

Chapter 1

Introduction

When developing software, one commonly relies on software libraries written by other developers. To avoid introducing defects into the software, one has to follow rules stated by the library developer. This includes the syntax and semantics of operations provided by the library. In a concurrent environment, a new set of problems related to the proper synchronization of threads is introduced.

Contracts for concurrency enable library developers to define restrictions on the usage of the library in a concurrent environment. In its basic form, it specifies which method sequences must be executed atomically. There are two extensions for contracts for concurrency. *Parametric contracts* allow to better identify methods that need to be executed atomically. Contracts with *spoilers* allow finer control over which thread interleavings violate the contract. To verify that a program satisfies the restrictions given by contracts for concurrency, one may use either static or dynamic analysis, both providing different advantages.

The main goal of this thesis is to design a dynamic analyzer that detects violations of parametric contracts with spoilers in programs written in the Java programming language. The analyzer is built using the *RoadRunner* framework. RoadRunner instruments programs under analysis and reports actions taken by the program via a simple interface. The proposed analyzer extends the instrumentation done by RoadRunner to extract additional information about the program under analysis. Apart from the analyzer itself, a parser for contract definitions is created.

The thesis is structured as follows. Chapter 2 describes the specifics of multi-threaded programming in Java, the Java memory model, and an overview of software errors related to concurrency. Approaches to the dynamic analysis of Java programs and instrumentation techniques are described. Two important frameworks are presented, the ASM framework for byte code instrumentation, and the RoadRunner framework for writing dynamic analyzers. Chapter 3 introduces contracts for concurrency, their modified versions, and a method for dynamic detection of contract violations. In Chapter 4, a dynamic analyzer for contracts is designed. Chapter 5 provides implementation details and testing approaches.

Chapter 2

Dynamic Analysis of Multi-threaded Programs in Java

This chapter focuses on a dynamic analysis that detects errors related to improper synchronization between threads in multi-threaded Java programs. In the first section, dynamic analysis is compared with other approaches to software verification. Then the most common types of errors found in multi-threaded programs are presented. The following section explains the basics of multi-threaded programming in Java. Then the most important concepts from the Java memory model are described.

The second part of this chapter deals with the techniques used for dynamic analysis of Java programs. The ASM framework for Java bytecode manipulation is introduced along with brief overview of the Java virtual machine. Finally, the RoadRunner framework is described in detail, as it is the basis for implementation of the contract analyzer.

2.1 Approaches to Software Verification

The goal of software verification is to make sure that the software meets all requirements [7]. There are several approaches to software verification, each of them having its own advantages and disadvantages. This section provides a summary of testing, dynamic and static analysis, abstract interpretation, theorem proving, and model checking.

Testing *Testing* consists of running the software under different conditions and checking the results of the computation (or observing other behavior of the software). To gain enough confidence that the software operates correctly in all conditions, a suitable set of *test cases* must be found, which is difficult, and sometimes impossible. Testing is best suited for confirming the presence of defects in software, not for proving their absence [7].

An important property of test cases is their *repeatability*, meaning that a certain test case will always yield the same result. When testing multi-threaded programs, this property does not hold because of the nondeterminism introduced by the thread scheduler. Threads are interleaved differently on each execution which means that errors may or may not appear. This makes discovering defects in multi-threaded programs difficult.

Dynamic Analysis *Dynamic analysis* works with information gathered during an execution of a program. The information may be analyzed during program execution (*on-the-fly* analysis) or at the end (*post-mortem* analysis). Even though the analysis works with in-

formation from a single execution, it can in some cases find errors that were not observed during the execution but may demonstrate themselves in similar executions [10]. The dynamic analysis also suffers from nondeterministic scheduling. The program under analysis may also behave differently due to being observed by the analyzer.

The analyzer proposed in this thesis performs on-the-fly dynamic analysis of contracts for concurrency and detects contract violations that occurred not only in the given run but also those that may have occurred in similar runs.

Static Analysis *Static analysis* is performed at compile time and it does not require the program to be running. The analysis is theoretically able to cover all possible executions of a program. In practice, it is limited by the fact that the number of thread interleavings in multi-threaded programs grows exponentially [10].

Abstract Interpretation *Abstract interpretation* takes the source code and symbolically executes it line by line, approximating the semantics of the program without performing all the calculations. It suffers from similar problems as static analysis.

Model Checking *Model checking* is a technique for checking whether a system satisfies certain correctness specification [10]. It is based on systematic or heuristic exploration of the state space. The drawback of this technique is that the state space of the program model can be huge.

Theorem Proving *Theorem proving* is a semi-automated approach to proving that certain facts are satisfied in the system. It is based on assumptions and general theorems about the system and uses mathematical reasoning [7].

2.2 Safety Errors in Multi-threaded Programs

Contracts for concurrency specify rules on using a set of methods in a concurrent setting. They aim at discovering errors specific to a concurrent environment. When compared to single-threaded programs, multi-threaded programs may encounter a whole new class of errors related to memory sharing between threads. Errors presented in this section are classified as *safety errors* in [10] as these are usually checked in various dynamic analyses.

Data Race A *data race* occurs when there are two unsynchronized accesses to a shared variable and at least one of them is a write access.

Atomicity Violation When a code block is required to be atomic, all program executions must be equivalent to an execution where the block is executed serially. Contract for concurrency primarily focus on atomicity violations [3].

Order Violation When certain operations are required to be executed in a certain order, and the order is not met in a given program execution, an *order violation* occurs. Contracts for concurrency can also detect order violations [3].

Deadlock General definition of a deadlock is presented in [10]. A program state contains a set S of deadlocked threads if, and only if each thread in S is blocked and waiting for some event that could unblock it, but such an event could only be generated by a thread from S.

Missed Signal A *missed signal* is present in a program execution when one or more threads are waiting for a signal, and the signal is never delivered.

2.3 Multi-threaded Programming in Java

Java provides built-in support for multi-threaded programming. This section describes a typical thread life cycle, synchronization of threads, and inter-thread communication, as these are important in dynamic analysis using contracts for concurrency.

A thread in Java is represented by a `Thread` instance. There are two ways to create a thread: by extending the `Thread` class, or by implementing the `Runnable` interface. Both approaches produce a `Thread` instance that executes the `run` method in a new thread when started.

To start a thread, the `start` method must be called (which will in turn call the `run` method). The thread will terminate upon returning from the `run` method. The `join` method is used in other threads to wait for a thread to terminate [12]. Listing 2.1 shows a thread creation example by extending the `Thread` class, Listing 2.2 shows the same example achieved by implementing the `Runnable` interface.

```
class MyThread extends Thread {
  @Override
  public void run() {
    System.out.println("This is executed in a new thread.");
  }

  public static void main(String args[]) {
    MyThread t = new MyThread();
    t.start();
    t.join();
  }
}
```

Listing 2.1: A simple program that creates a thread by extending the `Thread` class.

When accessing a shared resource from multiple threads, proper synchronization is usually required. In Java, every object gets an implicit monitor, which can be owned by only one thread at a given time. To enter the monitor, one must use either synchronized methods or synchronized statements. *Synchronized statements* are code blocks with an explicitly specified object whose monitor is entered before executing the block. *Synchronized methods* enter the monitor of the instance they are called upon [12]. Listing 2.3 shows examples of synchronized blocks and synchronized methods.

Communication between threads is achieved using the following methods: `wait`, `notify`, and `notifyAll`. All methods must be called within a synchronized context. Calling `wait` will suspend the calling thread until some other thread enters the same monitor and calls either `notify` or `notifyAll` [12].

```
class MyRunnable implements Runnable {
  public void run() {
    System.out.println("This is executed in a new thread.");
  }

  public static void main(String args[]) {
    Thread t = new Thread(new MyRunnable());
    t.start();
    t.join();
  }
}
```

Listing 2.2: A simple program that creates a thread by implementing the `Runnable` interface.

```
class Example {
  private int a = 0;

  public synchronized void inc1() {
    a++;
  }

  public void inc2() {
    synchronized (this) {
      a++;
    }
  }
}
```

Listing 2.3: A program with synchronized methods and statements. The `inc1` method is *synchronized*, on each call, the `Example` instance's monitor is entered. The `inc2` method is not synchronized but contains a *synchronized block* with an explicitly specified monitor (`this`).

Multi-threaded programs may use the `volatile` type modifier. It tells the compiler that the variable may be modified outside of the current thread.

2.4 Java Memory Model

Java memory model describes how threads in Java interact with each other using shared memory. The model defines several relations that are used by the dynamic analysis of contracts for concurrency, most notably the *happens-before* relation and the *synchronizes-with* relation.

Java memory model takes a program and an execution trace, and for each read operation decides if it is valid or not. The decision depends on the write operation that modified the data before the read operation. The compiler, runtime, and hardware must ensure that all executions of a program produce execution traces that are valid according to the model [8].

In a single-threaded program, it is only required that the program produces the same result as if it was run serially. The compiler is free to reorder instructions when it does

not affect the result of the computation. In multi-threaded programs, the reordering of instructions has to be limited when the threads interact with each other.

In the model, only certain program *actions* are considered. There are several orders defined over the actions which are used by the dynamic contract analysis: program order, synchronization order, and happens-before order.

The actions can be either intra- or inter-thread. An *inter-thread* action can be detected or influenced by another thread. An *intra-thread* action is for example adding two local variables and it is not important to the model. Nonvolatile reading or writing of a shared variable is an inter-thread action. *Synchronization actions* are inter-thread actions that include volatile reading or writing of variables, locking and unlocking of monitors, and starting and stopping of a thread [8]. Listing 2.4 shows examples of different kinds of actions.

```
class MySharedData {
  int mySharedVar = 0;

  public synchronized void MyMethod() {
    // synchronization action (entering a monitor)
    // intra-thread action (writing a local variable)
    int a = 42;
    // 2 inter-thread actions (reading and writing a shared variable)
    mySharedVar += a;
    // synchronization action (leaving a monitor)
  }
}
```

Listing 2.4: Various program actions classified from the Java memory model point of view. Entering and leaving `MyMethod` produces *synchronization actions*. Accessing `mySharedVar` is considered as an *inter-thread* action, but not as a synchronization action because `mySharedVar` is not declared as `volatile`.

Program order is a total order over all inter-thread actions from a given thread. It reflects the order in which these actions would be executed if run by the intra-thread semantics.

Synchronization order is a total order over all synchronization actions of an execution. Within each thread, the synchronization order is consistent with the program order. The *synchronized-with* relation is defined on certain actions. For example: starting a thread is *synchronized-with* the first action in the new thread.

Happens-before order is a partial order. If an action *happens-before* another, the first action is visible to and ordered before the second action. If actions x and y belong to the same thread and x comes before y in program order, then x *happens-before* y. If x *synchronizes-with* y, then x *happens-before* y. Figures 2.1 and 2.2 illustrates the *happens-before* relation in simple programs.

A *data race* occurs, when there are two accesses to the same variable, at least one of which is write, and these accesses are not ordered by *happens-before* [8]. This situation is illustrated in Figure 2.2.

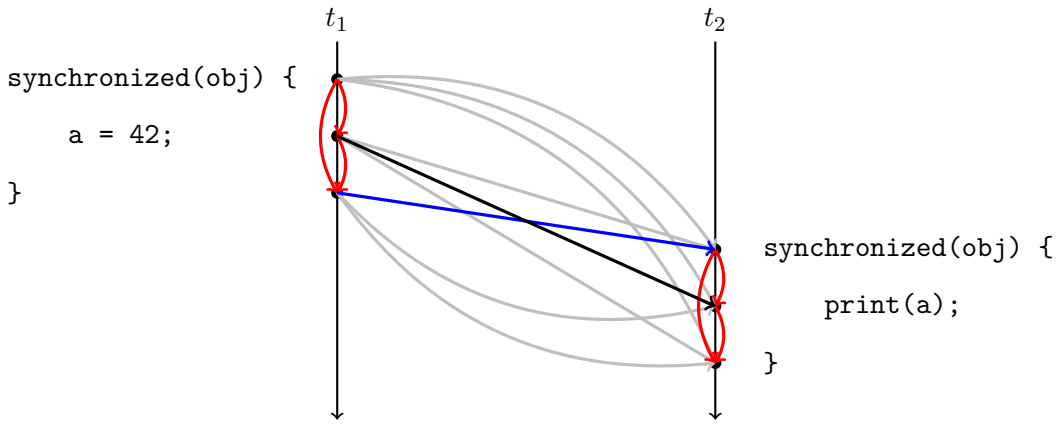


Figure 2.1: *Happens-before* relations in a correctly synchronized program consisting of threads t_1 and t_2. Each arrow represents a *happens-before* relation. The red arrows represent the program order, the blue arrow represents the *synchronizes-with* relation. Grey arrows complete the transitive closure. The conflicting accesses to variable `a` are not data races, because they are ordered by *happens-before* (the black arrow).

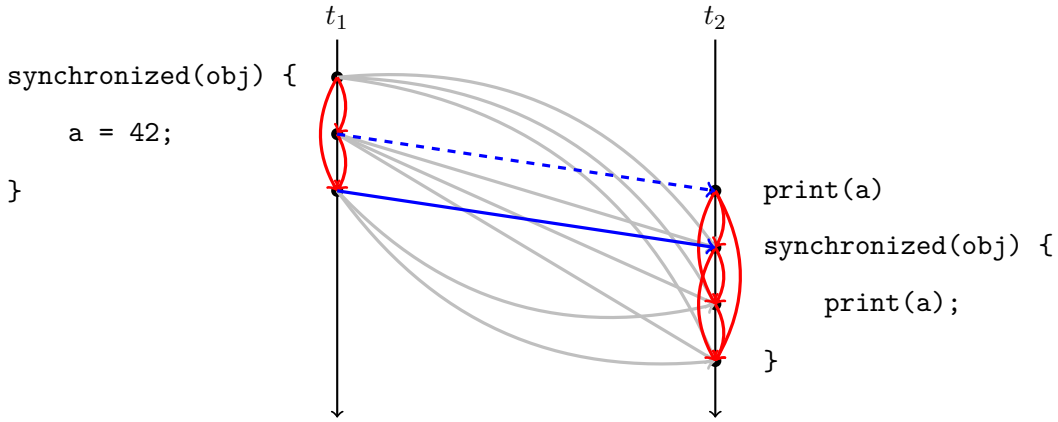


Figure 2.2: *Happens-before* relations in an incorrectly synchronized program (each solid arrow represents a *happens-before* relation). There is no *happens-before* relation between conflicting accesses `a=42` and `print(a)` (the dashed line), creating a data race.

2.5 Instrumentation of Java Bytecode

Instrumentation is the act of inserting instructions into an existing program to extract useful information at runtime. Instrumentation can be used to measure performance, log events, or perform dynamic analysis. The running program should not be aware that it is being instrumented and the result of the computation should remain the same. Instrumentation may add significant overhead to the program. For example, programs instrumented by the RoadRunner framework are roughly ten times slower [6].

In Java, the instrumentation is done by changing the bytecode. There are several general-purpose frameworks for modifying the Java bytecode. In this section, the ASM framework is described as it is used by the RoadRunner framework, which is the basis of this Master's thesis.

2.5.1 Java Bytecode Overview

Programs written in Java are compiled into Java bytecode which is executed by the Java Virtual Machine. Every class gets compiled into a Java class file containing the following sections [2]:

- A section with information about the class itself, such as the name of the class, the super class, implemented interfaces, and class annotations.
- One section per field, containing the field name, type, modifiers, and annotations.
- One section per method (and constructor), containing the name of the method, the return type, type of parameters, annotations, and compiled code of the method.

Java class files also contain a *constant pool* section that holds all numeric, type, and string constants which are then referenced from other sections of the file. The whole structure is shown in Table 2.1. The Java class file format is described in detail in the Java Virtual Machine Specification [11].

Modifiers, name, super class, interfaces	
Constant pool	
Annotations	
Attributes	
Fields	Modifiers, name, type Annotations Attributes
Methods	Modifiers, name, return and parameter types Annotations Attributes Code

Table 2.1: Structure of the Java class file. Adapted from [2], simplified.

The Java Virtual Machine operates on two kinds of types: *primitive types* and *reference types*. Examples of primitive types are `int`, `long`, `boolean`, or `double`. There are three kinds of reference types: class types, array types, and interface types. The array type consists of a component type which can also be an array type. For example, `int[]` represents an array type with component type of `int`. All reference types may hold a special null reference, which is also the default value of reference types.

Compiled classes do not contain any `package` or `import` statements, so all type names must be fully qualified. Internally, class files use slashes instead of dots in type names, so for example `java.lang.Object` becomes `java/lang/Object`. In most places, Java types are represented with *type descriptors*. Each primitive type is assigned a single character: `I` for `int`, `D` for `Double`, and so on. Classes and interfaces are written with prefix `L` and semicolon at the end, so `String` becomes `Ljava/lang/String;`. Arrays are represented using a `[` and the element type, so an array of integers is `[I`, an array of strings is `[Ljava/lang/String;`. Similarly, *method descriptors* are used to represent the return type of a method and types of all method parameters. For example, a method declared as `double m(int i, String s)` would be represented as `(ILjava/lang/String;)D`. In method descriptors, `V` is used when the method returns `void`.

When executing, on each method invocation, the Java Virtual Machine creates a new *frame*. Each frame contains its own local variables and an operand stack. When the method invocation is completed, the frame is destroyed.

Local variables are addressed by indexing. Each variable can hold a single value of a primitive or reference type with the exception of `long` and `double` which require a pair of variables. At index 0, there is a reference to the object the method was invoked on (the value of `this` in Java). Class methods (marked as `static` in Java) do not use this index. Starting at index 1 (or 0 in case of class methods), method parameters are stored. After the parameters, local variables may be stored.

Each frame contains an *operand stack*, which is initially empty. Various instructions are used to load values onto the stack, either from local variables or fields. Other instructions take operands from the stack and push the result back. When calling other methods, the parameters are also prepared on the stack.

Java Virtual Machine instructions can be divided into several categories. Load and store instructions move values between local variables and the operand stack. For example, the `iload_3` instruction pushes the value (which is of type `int`) from the local variable at index 3 to the operand stack. Arithmetic instructions usually take two values from the operand stack, compute the result, and store it back on the stack. For example, the `fmul` instruction will multiply two values of type `float`. Type conversion instructions convert the value on the top of the stack. Control transfer instructions, such as `ifeq` or `goto`, cause the execution of instruction other then the immediately following.

To create new arrays and objects, instructions `new`, `newarray`, and `anewarray` are used. Methods are invoked using these five instructions: `invokevirtual`, `invokeinterface`, `invokespecial`, `invokestatic`, and `invokedynamic`, each used in slightly different circumstances. Exceptions are thrown using the `athrow` instruction. Entering a monitor is achieved by `monitorenter` and `monitorexit` instructions, which are used by synchronized statements in Java. An example of a method represented by bytecode is shown in Listing 2.5.

2.5.2 The ASM framework

The ASM framework allows generating and modifying Java classes directly in bytecode. It can be used both statically (for example during compilation) or dynamically (to create classes at runtime). The ASM framework provides an interface for loading and storing the bytecode using higher-level abstractions, such as constants, identifiers, methods, fields, and others [2].

There are two interfaces available: the *core API* with an *event-based* representation of classes, and the *tree API* with an *object-based* representation. The core API processes classes sequentially. When parsing a class, the ASM parser will produce an event for each element of the class. When writing a class, the writer creates the class based on a sequence of events. The tree API loads the whole class and creates a tree of objects representing the class. The core API is faster and requires less memory, however, it is not practical for complex transformations [2]. The RoadRunner framework uses the core API.

The core API is based on the `ClassVisitor` abstract class. The class contains methods for visiting different sections of a class, for example, `visitAttribute`, `visitMethod`, or `visitField`. Complex sections, such as methods or fields, have their visitor classes. For example, the `MethodVisitor` class contains methods such as `visitLocalVariable`, `visitCode`, or `visitParameter` [2].

```
public void foo(java.io.FileWriter, int, int)
  descriptor: (Ljava/io/FileWriter;II)V
  flags: (0x0001) ACC_PUBLIC
  Code:
    stack=2, locals=5, args_size=4
       0: iload_2
       1: iload_3
       2: iadd
       3: istore        4
       5: aload_1
       6: iload         4
       8: invokevirtual #2    // Method java/io/FileWriter.write:(I)V
      11: aload_1
      12: invokevirtual #3    // Method java/io/FileWriter.close:()V
      15: return
```

Listing 2.5: An example of a method bytecode viewed using the `javap` command. The method takes three parameters: a file writer and two integers. There are 5 local variables: the object the method was called on (index 0), method parameters (indexes 1–3), and a local variable (index 5). On lines 0–3, the two integers are loaded on to the operand stack, added together, and the result is stored in a local variable. Lines 5–7 calls the `write` method on the file writer, lines 11–12 calls the `close` method. Operands on lines 8 and 12 are indexes to the constant pool section.

To generate a new class, one has to create a `ClassWriter` instance, which is a subclass of `ClassVisitor`. Then a sequence of visit methods must be called, such as `visitField` or `visitMethod`. The `ClassWriter` instance will generate appropriate bytecode on each call.

To read and parse a class, one has to create a `ClassReader` instance. The reader will produce a sequence of events for each section of the class. To consume those events, a `ClassVisitor` instance must be given to the reader. The reader will then call appropriate visit methods on the visitor as it is parsing the class. To demonstrate this, one can create a `ClassReader` and connect it to a `ClassWriter` (which is a subclass of `ClassVisitor`). The reader will call visit methods on the writer, effectively copying the class. The typical class transformation is shown in Figure 2.3.

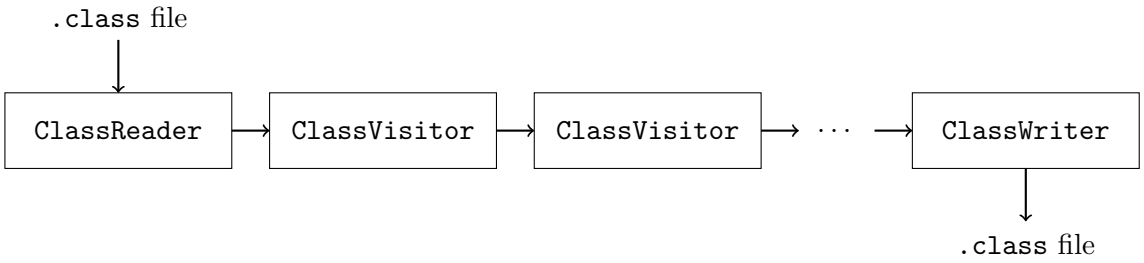


Figure 2.3: The typical architecture for a class transformation using the ASM framework. A `ClassReader` instance reads the class, then one or more `ClassVisitor` instances modify the class, and then a `ClassWriter` instance writes the modified class back to a file.

2.6 Dynamic Analysis using RoadRunner

The RoadRunner framework is used for the dynamic analysis of concurrent programs written in Java. RoadRunner instruments programs to obtain a stream of events that are useful for dynamic analysis, such as memory accesses, synchronizing on a lock, forking or joining of threads, and so on. This event stream is then available to various analysis tools. Multiple tools can be chained together, each tool acting as a filter over the events. This allows complex analyses to be built from simpler, modular tools [6].

RoadRunner aims to simplify writing dynamic analysis tools. A RoadRunner analysis tool only needs to handle events of interest. RoadRunner will ensure that the event is properly detected and the event handler is called. To store the state of the analysis, RoadRunner provides support for associating data with memory locations, locks, or threads.

2.6.1 The RoadRunner Programming Interface

Every analyzer in RoadRunner is based on the `Tool` class. Listing 2.6 contains the most important methods of `Tool`. During the analysis, every time an action is detected, the appropriate method in `Tool` is called, along with an `Event` object that contains information about the event. The following events are detected by the RoadRunner framework:

- method entry and exit,
- memory accesses (reads and writes to fields and variables),
- lock acquires and releases,
- synchronization signals (wait and notify),
- thread forking and joining.

There are several subclasses of the `Event` class with specific information about events.

```
public abstract class Tool {
  // event handlers for accessing a memory location
  public void access(AccessEvent fae) { }
  public void volatileAccess(VolatileAccessEvent fae) { }
  // event handlers for entering and exiting methods
  public void enter(MethodEvent me) { }
  public void exit(MethodEvent me) { }
  // event handlers for locking
  public void acquire(AcquireEvent ae) { }
  public void release(ReleaseEvent re) { }
  // event handlers for thread events
  public void preJoin(JoinEvent je) { }
  public void postJoin(JoinEvent je) { }
  public void preStart(StartEvent se) { }
  public void postStart(StartEvent se) { }
  // shadow location initialization
  public ShadowVar makeShadowVar(AccessEvent ae) { }
}
```

Listing 2.6: The abstract class `Tool`. Only selected public methods are shown.

RoadRunner allows associating data with objects from the program under analysis. For each thread, a `ShadowThread` object is created which contains a reference to the underlying thread. Similarly, for each lock, a `ShadowLock` object is created. Both extend the `Decoratable` class that allows storing of arbitrary information. For associating data with memory locations, a *shadow location* is created when the location is first accessed.

Multiple tools can be chained together. Each event handler method forwards the `Event` instance to the next tool in the chain by default. If the event is not forwarded, the tool becomes a filter over the event stream. This can be used to filter out events that are not interesting to a particular analysis and then performing the analysis in the next tool [6].

2.6.2 RoadRunner Synchronization Models

In RoadRunner, all threads of the program under analysis generate events. The events are also handled by the same thread that generated them which means that several event handlers may be running concurrently. Tools written for RoadRunner must provide internal synchronization to ensure that no concurrency-related errors occur in the tool itself. RoadRunner contains an option to serialize all events. In this mode, there is only one event handler running at a time [6].

2.6.3 Instrumentation Performed by RoadRunner

RoadRunner uses a modified version of the ASM framework to instrument the program under analysis. Before a class is loaded, it is instrumented. The instrumented code will then produce events that will be sent to the tool chain for an analysis. Three important kinds of actions are instrumented: field accesses, method invocations, and monitor entries and exits.

Field accesses are instrumented by adding two new methods for each field: one for reading and one for writing to the field. In these methods, write and read events are generated. In the rest of the code, all `getfield` and `putfield` instructions are replaced with calls to the corresponding access methods. RoadRunner allows tools to store arbitrary data related to a field in *shadow variables*. For each field, a new field of the `ShadowVar` type is created to store the data. Listings 2.7 and 2.8 shows a simple class before and after field instrumentation.

```
private int bar;

public void foo();
   0: aload_0
   1: aload_0
   2: getfield      #2    // Field bar:I
   5: bipush        42
   7: iadd
   8: putfield      #2    // Field bar:I
  11: return
```

Listing 2.7: An example class bytecode viewed using the `javap` tool, simplified.

Method invocations are tracked by creating a wrapper method for each method. The original method is renamed, but otherwise left intact (the code may however be further instrumented to obtain other information, such as field accesses). Then a wrapper method

```
public int bar;

public transient rr.state.ShadowVar __$rr_bar;

public void __$rr_put_bar(int, int, rr.state.ShadowThread)
  (code omitted)

public int __$rr_get_bar(int, rr.state.ShadowThread)
  (code omitted)

public void foo();
   0: invokestatic  #51 // Method rr/state/ShadowThread
                        // .getCurrentShadowThread:()Lrr/state/ShadowThread;
   3: astore_2
   4: aload_0
   5: aload_0
   6: iconst_1
   7: aload_2
   8: invokespecial #56 // Method __$rr_get_bar:(ILrr/state/ShadowThread;)I
  11: bipush        42
  13: iadd
  14: iconst_2
  15: aload_2
  16: invokespecial #53 // Method __$rr_put_bar:(IILrr/state/ShadowThread;)V
  19: return
```

Listing 2.8: Code from Listing 2.7 instrumented by RoadRunner. For the `bar` field, two access methods are added and a new field of type `ShadowVar`. In the `foo` method, the `bar` field is accessed using methods `__$rr_get_bar` and `__$rr_put_bar`. These methods take the current shadow thread as an argument which is obtained by calling `getCurrentShadowThread`.

with the same name as the original one is created. The wrapper method generates enter and exit events. In order to detect abnormal method exits that are caused by an exception being thrown, the call to the original method is wrapped in a try block. When an exception is caught, the exit event is generated and the exception is re-thrown. An example of a method instrumented by RoadRunner is shown in Listing 2.9.

Monitor entries and exits are handled differently for synchronized statements and synchronized methods. Synchronized statements in Java are represented by `monitorenter` and `monitorexit` instructions. RoadRunner extends all occurrences of these instructions with calls to methods that generate acquire and release events. Synchronized methods in Java do not need `monitorenter` and `monitorexit` instructions, the locking is performed implicitly by the Java Virtual Machine. In RoadRunner, synchronized methods are replaced with synchronized statements that are then instrumented as described above. For each synchronized method, a wrapper method is created. The original method's synchronized flag is cleared. The wrapper method, which is also not synchronized, contains a synchronized statement with call to the original method. Synchronized methods are in the end wrapped twice, the first wrapper generates synchronization events and the second one generates method invocation events.

```
public int __$rr_foo__$rr__Original_(int);
   0: invokestatic  #20 // Method rr/state/ShadowThread
                        // .getCurrentShadowThread:()Lrr/state/ShadowThread;
   3: astore_3
   4: iload_1
   5: ireturn

public int foo(int);
   0: invokestatic  #20 // Method rr/state/ShadowThread
                        // .getCurrentShadowThread:()Lrr/state/ShadowThread;
   3: astore_3
   4: aload_0
   5: sipush        508
   8: aload_3
   9: invokestatic  #27 // Method rr/tool/RREventGenerator
                        // .enter:(Ljava/lang/Object;ILrr/state/ShadowThread;)V
  12: aload_0
  13: iload_1
  14: invokespecial #29 // Method __$rr_foo__$rr__Original_:(I)I
  17: aload_3
  18: invokestatic  #33 // Method rr/tool/RREventGenerator
                        // .exit:(Lrr/state/ShadowThread;)V
  21: goto          29
  24: aload_3
  25: invokestatic  #33 // Method rr/tool/RREventGenerator
                        // .exit:(Lrr/state/ShadowThread;)V
  28: athrow
  29: ireturn
```

Listing 2.9: Method `int foo(int a)` instrumented by RoadRunner. The original method was renamed to `__$rr_foo__$rr__Original_` and a new method with the original name was created. This method generates enter and exit events and calls the original method.

Chapter 3

Contracts for Concurrency

When developing software, one frequently uses modules created by someone else via its programming interface. For example, in object-oriented programming, the interface consists of public methods of a given class. Accessing the interface requires one to follow a protocol consisting of: (i) syntax, i.e. types of parameters and return values, (ii) semantics, i.e. the expected behavior for given input parameters, and (iii) access restrictions. Access restrictions include the domain of valid values, dependencies on other services, and atomicity violations [3].

Contracts for concurrency [4], [13], are a case of a software protocol that expresses access restrictions in a concurrent setting. In its basic form, they specify sequences of methods that must be executed atomically. Contracts for concurrency help detect high-level data races in a program. A *high-level* data race occurs on a higher abstraction layer. Program that is free of data races as defined by the Java memory model can still contain high-level data races when modifying complex data structures [1]. As an example, consider an object that represents a pair of coordinates with two synchronized methods: `setX` and `setY`. Even though both methods are executed atomically, there is a window between setting the first and the second coordinate where the object is in an inconsistent state, allowing for a high-level data race.

The contracts can be extended with parameters to reflect the data flow between the methods (so that only methods manipulating the same data must be executed atomically). Another extension adds so-called *spoilers* (so that given sequence must be executed atomically only with respect to only certain sequences). Both extensions can be combined. This chapter defines basic contracts, as well as both extensions to them. Then a method for dynamic validation of contracts for concurrency is presented. The analyzer, implemented in this thesis, is based on this method.

3.1 Basic Contracts

A *contract* is formally defined in [4] as follows. Let $\Sigma_{\mathbb{M}}$ be a set of all public method names (the API) of a module or a library. A *contract* is a set $\mathbb{R}$ of *clauses*. Each clause $\varrho \in \mathbb{R}$ is a regular expression over $\Sigma_{\mathbb{M}}$. A contract violation occurs when any of the sequences in a contract is interleaved with an execution of a method from $\Sigma_{\mathbb{M}}$ over the same object.

Example. Consider a map implementation with the following operations: `put(key, value)`, `get(key)`, `remove(key)`, and `contains(key)`. Then a contract for this class may contain the following clauses:

(ϱ_1) `put get`
(ϱ_2) `contains (put|get|remove)`

Clause ϱ_1 states that when an element is put into the map and then retrieved, it should be executed atomically (because the element may be removed between the calls). Clause ϱ_2 states that when the program modifies the map based on the result of the `contains` call, it should be atomic.

3.2 Parametric Contracts

In some situations, the definition of contracts may be too restrictive, producing false alarms. In [3], contracts are extended with parameters to reflect the data flow between methods. Consider the following example:

```
if (q.contains(42)) q.remove(42);
```

These two calls must be executed atomically only if they share the same argument. This dependency can be expressed using *meta-variables* placed as the parameters or return values of methods. Parameters that should not be taken into account are marked with free meta-variable (denoted with an underscore).

Example. The example from Section 3.1 can be extended with parameters:

(ϱ_1) `put(X,_) _=get(X)`
(ϱ_2) `_=contains(X) ( put(X,_) | _=get(X) | remove(X) )`

Clause ϱ_1 cares about calls to `put` and `get` that operate on the same key (the `X` meta-variable) but it is not concerned with the value that is put or retrieved (the `_` meta-variable). Similarly, in clause ϱ_2, only method calls operating with the same key must be atomic.

The basic definition of contracts contains one implicit parameter, the object that the method was called upon (`this` in Java) [4]. The atomicity is required only on methods called upon the same object (as these method calls usually modify the same data). To better illustrate this, the example can be rewritten as:

(ϱ_1) `X.put(Y,_) _=X.get(Y)`
(ϱ_2) `_=X.contains(Y) ( X.put(Y,_) | _=X.get(Y) | X.remove(Y) )`

3.3 Contracts with Spoilers

In [3], contracts are extended with contextual information to distinguish which method sequences violate the contract. Each clause of the basic contract is called a *target* and is assigned a set of so-called *spoilers*. A spoiler is a set of method sequences that may violate its target.

Consider clause ϱ_1 from the example in Section 3.1. If the element that was put into the map is concurrently removed or updated before the `get` call, a contract violation should be

detected. However, calling `contains` or `get` on the element will not affect the computation and should not be marked as a contract violation. In this example, methods `put` and `remove` are spoilers for a target ϱ_1, denoted as `put get` $\rightsquigarrow$ `put|remove`.

Formally, as defined in [3], let $\mathbb{R}$ be the set of *target* clauses where each target $\varrho \in \mathbb{R}$ is a regular expression over $\Sigma_{\mathbb{M}}$. Let $\mathbb{S}$ be the set of *spoilers* where each spoiler $\sigma \in \mathbb{S}$ is a regular expression over $\Sigma_{\mathbb{M}}$. A *contract* is a relation $\mathbb{C} \subseteq \mathbb{R} \times \mathbb{S}$ defining for each target, which spoilers may cause atomicity violation.

Contract violation is observed when a target sequence $\varrho \in \mathbb{R}$ is fully interleaved by a spoiler sequence $\sigma \in \mathbb{C}(\varrho)$ and the sequences are executed on the same object.

Example. The example from section 3.1 can be extended with spoilers:

(ϱ_1) `put get` $\rightsquigarrow$ `put|remove`
(ϱ_2) `contains (put|get|remove)` $\rightsquigarrow$ `put|remove`

When combining parametric contracts with spoilers, the spoilers may also contain parameters. Then a contract violation is detected only when spoiler arguments match target arguments.

Example. Examples from sections 3.2 and 3.3 combined together:

(ϱ_1) `X.put(Y,_) _ = X.get(Y)` $\rightsquigarrow$ `X.put(Y,_)|X.remove(Y)`
(ϱ_2) `_ = X.contains(Y) ( X.put(Y,_) | _ = X.get(Y) | X.remove(Y) )`
$\rightsquigarrow$ `X.put(Y,_)|X.remove(Y)`

3.4 Dynamic Contract Validation

In [3], a dynamic contract validation method is proposed for contracts with spoilers. Parametric contracts are not included in the method. This section provides an overview of the method. The analyzer designed in Chapter 4 uses this method and extends it with parameters.

3.4.1 Multi-threaded Program Traces

In the context of the dynamic on-the-fly contract validation, multi-threaded *program trace* consists of events of the following types:

- thread forking or joining another thread,
- thread entering or exiting a method,
- thread acquiring or releasing a lock.

All events in a trace are indexed by their position in the trace. Let $\mathbb{T}$ be a set of threads, $\mathbb{R}$ a set of targets, $\mathbb{S}$ a set of spoilers, $\mathbb{C} \subseteq \mathbb{R} \times \mathbb{S}$ a set of contracts, and $\mathbb{L}$ a set of locks. The set of all events that can be generated by a thread $t \in \mathbb{T}$ is then denoted as $\mathbb{E}_t$. Let $\mathbb{E} = \bigcup_{t \in \mathbb{T}} \mathbb{E}_t$. A *trace* is then a sequence $\tau = e_1 \dots e_n \in \mathbb{E}^+$ [3].

Given a trace $\tau = e_1 \dots e_n \in \mathbb{E}^+$, we call its subsequence $r = e_{i_1} e_{i_2} \dots e_{i_k}$, $1 < k \leqslant n$, an *instance* of a target $\varrho \in \mathbb{R}$ if, and only if:

1. r consists of well-paired method enter and exit events,

2. all enter events of r match the regular expression of ϱ,

3. apart from events $e_{i_1}, \ldots, e_{i_k}$, there is no event from the alphabet of ϱ executed by t between events e_{i_1} and e_{i_k}.

Example. Given target $\varrho = abc$, and a trace $\tau_1 = adbdc$, there is a target instance $r = e_1 e_3 e_4$. In trace $\tau_2 = acbdc$, there is no target instance.

A *spoiler instance* s of a spoiler $\sigma \in \mathbb{S}$ is defined similarly. We let $start(r) = e_{i_1}$ and $end(r) = e_{i_k}$ denote the first and last events of a target, respectively. Likewise, $start(s)$ and $end(s)$ denote the first and last events of a spoiler, respectively [3].

3.4.2 Contract Violation

A contract is violated when there is a target instance that is fully interleaved with a spoiler instance from another thread. The interleaving is defined using a *happens-before* relation, which is in the context of contracts defined as follows [3]. A *happens-before relation* $\prec_{hb}$ over a trace $\tau = e_1 \ldots e_n \in \mathbb{E}^+$ is the smallest transitively closed relation on the set of events from τ such that $e_j \prec_{hb} e_k$ holds when $j < k$ and one of the following holds:

1. both e_j and e_k are executed by the same thread,

2. both e_j and e_k acquire or release the same lock,

3. one of e_j and e_k is a fork or a join performed by thread t, and the other is executed by thread u.

A contract $(\varrho, \sigma) \in \mathbb{C}$ is *violated* in a trace τ if, and only if there is a target instance r in the trace and a spoiler instance s in the trace such that:

$$start(s) \not\prec_{hb} start(r) \wedge end(r) \not\prec_{hb} end(s)$$

The violation occurs when the spoiler may have started after the target started and it may have ended before the target ended. When given a complete program execution trace, all target and spoiler instances can be detected, the happens-before relation can be deduced, and all contracts can be easily checked for violations. The trace, however, can get large and make this approach unpractical. For this reason, several optimizations are introduced in [3], which are presented in the next section. An example of a program trace containing a contract violation is shown in Figure 3.1.

3.4.3 On-the-fly Contract Validation

To check contract validations, it is not required to keep the entire program execution trace. A *trace window* is kept instead. Events are moved to the trace window as soon as they become available and are removed under certain conditions. The goal is to keep the window as small as possible.

Spoiler instances can be safely removed from the window whenever a contract violation that would be detected with the spoiler can be detected without it. A spoiler instance can be removed from the window whenever a newer instance of the same spoiler is detected [3].

A target instance r can be safely removed with respect to a spoiler instance s whenever a contract violation that would be detected between r and s, can be detected between s and

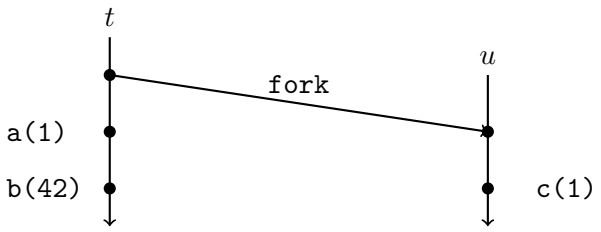


Figure 3.1: An example of a program trace containing a contract violation. Consider a target `a(X) b(_)` and a spoiler `c(X)`. In thread t, a target instance r is detected with $X = 1$. In thread u, a spoiler instance s is detected with $X = 1$. The parameters match in both instances. The instances are not synchronized, so $start(s) \not\vdash_{hb} start(r) \wedge end(r) \not\vdash_{hb} end(s)$ holds, which means that there is a contract violation.

another target instance too. Note that target instances may be removed only with respect to a given spoiler, not in general [3].

To further reduce the required information about the trace, *vector clocks* are used. Vector clocks are described in the next section. For each target and spoiler instance in the trace window, only vector clocks of their beginning and end need to be kept.

The method for on-the-fly contract validation does the following. At method entry events, target and spoiler sequences are detected. At method exit events, it is detected whether a target or a spoiler instance has ended. When a target instance ends, spoiler instances from the trace window are checked if they violate the target. When a spoiler instance ends, target instances from the trace window are checked if they are violated by the spoiler. At method exit, target and spoiler instances are also discarded when no longer needed [3].

3.4.4 Vector Clocks

The on-the-fly dynamic analysis of contracts uses vector clocks and the *happens-before* relation the same way it is used in the FastTrack algorithm [5]. A *vector clock* $VC : \mathbb{T} \to \mathbb{N}$ consists of clock values for each thread $t \in \mathbb{T}$. Vector clocks are partially ordered with $\sqsubseteq$, can be joined with $\sqcup$, and contain a minimal element $\bot_V$. The t-component of a vector clock is incremented using the inc_t function.

$$
\begin{aligned}
V_1 \sqsubseteq V_2 \quad \text{iff} \quad & \forall t.\, V_1(t) \leq V_2(t) \\
V_1 \sqcup V_2 &= \lambda t.\, max(V_1(t), V_2(t)) \\
\bot_V &= \lambda t.\, 0 \\
inc_t(V) &= \lambda u.\, \text{if } u = t \text{ then } V(u) + 1 \text{ else } V(u)
\end{aligned}
$$

Example. Consider threads t_1, t_2, t_3, and two vector clocks: $V_1 = \langle 1, 0, 2 \rangle$, $V_2 = \langle 1, 0, 5 \rangle$. Then $V_1 \sqsubseteq V_2$ is true, $V_1 \sqcup V_2 = \langle 1, 0, 5 \rangle$, and $inc_{t_2}(V_1) = \langle 1, 1, 2 \rangle$.

During the analysis, three kinds of clocks are kept. For each thread $t \in \mathbb{T}$, a vector clock $\mathbb{C}_t$ stores information about the last synchronization with other threads. For each lock $l \in \mathbb{L}$, a vector clock $\mathbb{L}_l$ holds information about the last thread that released the lock. For each event $e \in \tau$, a vector clock VC_e is kept [3].

The *happens-before* relation is defined using vector clocks. For an event e_t from a thread t and an event e_u from a thread u, $e_t \prec_{hb} e_u$ when $VC_{e_t}(t) \leq VC_{e_u}(t)$. The clocks are updated on the following actions:

- Fork — when a thread t creates a new thread u:

$$\mathbb{C}_u \leftarrow \mathbb{C}_u \sqcup \mathbb{C}_t$$
$$\mathbb{C}_t \leftarrow inc_t(\mathbb{C}_t)$$

  The new thread will get all *happens-before* relations from the parent thread. Then the parent thread is updated so that events coming after the fork will not *happen-before* events in the new thread.

- Join — when a thread t waits for a thread u to finish.

$$\mathbb{C}_t \leftarrow \mathbb{C}_t \sqcup \mathbb{C}_u$$
$$\mathbb{C}_u \leftarrow inc_u(\mathbb{C}_u)$$

  The thread that waits for the joining thread will get all *happens-before* relations from the joining thread. Then the joining thread is updated so that events coming after the join will not *happen-before* events in the waiting thread.

- Release — when a thread t releases a lock l.

$$\mathbb{L}_l \leftarrow \mathbb{C}_t$$
$$\mathbb{C}_t \leftarrow inc_t(\mathbb{C}_t)$$

  The releasing thread will be synchronized with the thread that will acquire the lock in the future. The thread does not know with which thread, so the thread's vector clock is stored in the lock. Then the thread is updated so that events coming after the release will not be synchronized with the thread that acquires the lock in the future.

- Acquire — when a thread t acquires a lock l.

$$\mathbb{C}_t \leftarrow \mathbb{C}_t \sqcup \mathbb{L}_l$$

  The acquiring thread will get *happens-before* relations from the lock which holds the vector clock from last release operation.

- Event clocks are set when an event enters the window trace. For an event $e \in \tau$ executed by a thread $t \in \mathbb{T}$:

$$VC_e \leftarrow \mathbb{C}_t$$

3.5 Previous Work

There are several existing implementation of dynamic analyzers for contracts for concurrency. In [4], the IBM Concurrency Testing Tool is used for tracking the basic contracts in Java programs. In [3], the ANaConDA framework is used for tracking parametric contracts with spoilers in programs written in C/C++. In [9], the RoadRunner framework is used to track parametric contracts with spoiler in Java programs. The prototype implementation in [9] served as a reference for this thesis.

Chapter 4

Design of a Dynamic Analyzer for Parametric Contracts with Spoilers

This chapter describes the proposed dynamic analyzer for parametric contracts with spoilers. The analyzer follows the method for dynamic analysis of contracts described in [3] and extends it to support parametric contracts.

The analyzer is built as a new tool for the RoadRunner framework. The input is a program under analysis and a contract definition. The analyzer is then able to detect contract violations in the program and report them. The RoadRunner framework was modified to support obtaining method arguments and return values.

Section 4.1 provides an architectural overview of the analyzer itself. Section 4.2 describes several restrictions that were placed on the analyzer in the design phase. In Section 4.3, necessary changes to RoadRunner itself are presented. Section 4.4 describes how a contract is defined and processed before the analysis is started. The core function of the analyzer is described in Section 4.5. Section 4.6 describes how the analyzer interacts with RoadRunner.

4.1 Overview of the Contract Analyzer

This section provides a high-level overview of the contract analyzer. The `ContractAnalyzer` class is the core of the analyzer. It receives events from the program under analysis, detects target and spoiler instances, and looks for contract violations. It manages data stored with threads and locks, such as trace windows and vector clocks. The `ContractAnalyzer` class can be instantiated without any dependencies from the RoadRunner project, which is useful for testing purposes. Section 4.5 describes `ContractAnalyzer` in detail.

The `ContractTool` class is a subclass of RoadRunner's `Tool` class. During the initialization of `ContractTool`, the contract definition file is parsed and `ContractAnalyzer` is created. In `ContractTool`, relevant methods are overridden to receive events from RoadRunner, such as lock acquire and release or method exit. These events are then processed and sent to `ContractAnalyzer`. Section 4.6 provides a detailed description of `ContractTool` and Section 4.4 describes the parsing and representation of contracts.

For each thread, a `Window` instance is created by `ContractAnalyzer`. It stores information about target and spoiler instances in a trace window. On method exit, existing instances are advanced, new instances are started, and for all finished instances, contract violations are checked.

4.2 Constraining Analyzer Capabilities

The analyzer is designed with several restrictions based on the previous work, such as [9] or [4], to improve its performance. First, a restriction is placed on the parameters in contract definitions to reduce the number of instances in a trace window. Then the conditions for removing instances from the trace window are discussed and a related contract restriction is introduced. Finally, it is described how nested method calls should be handled.

4.2.1 Avoiding Cloning of Target and Spoiler Instances

The method for analyzing contracts described in [9] produces an enormous number of instances being tracked at the same time. A lot of instances are created because of the necessity to clone target and spoiler instances before they are advanced. Consider the following target: `a(X) b(Y) c(X,Y)` and the following program trace: `a(1) b(2) b(3) c(1,3)`. When `a(1)` enters the trace window, a new instance is created and the value of `X` is set to 1. But when processing `b(2)`, the analyzer cannot reliably decide whether the method call belongs to the instance or not (there might be `c(1,2)` later in the trace). The only option is to keep the instance and create a duplicate instance which is then advanced (while setting `Y` to 2).

To prevent duplication of instances, the following restriction was put on the contract definition. All target and spoiler parameters must be assigned in the first call of a given target or spoiler. This ensures that there is no ambiguity in deciding whether a given method call advances an instance or not. For example, the target from the previous example is invalid because the value of `Y` remains unknown after the first method call.

4.2.2 Invalidating Instances

A target or spoiler instance, as defined in Chapter 3, requires that no method belonging to the alphabet of a given target or spoiler may be called between the events that form the instance. In practice, it means that a running instance must be discarded when a method belonging to the target or spoiler is called. For example, consider a target `abc` and the following program trace: `aa`. After the first `a`, a new instance is created. After accepting the second `a`, the instance must be discarded.

When tracking parametric contracts, instances cannot be easily discarded. Consider a running instance of a target (or a spoiler), all of its parameters are assigned a value. When a method is called that belongs to the alphabet of the instance's target, three kinds of situations can happen:

1. The method matches the target definition and method arguments match the values of instance parameters. The instance is advanced with the method.

2. The method matches the target definition but method arguments conflict with the values assigned to the instance. The instance cannot be advanced but it also should not be discarded. The method call most likely belongs to another instance.

3. The method does not match the target definition. According to the definition in Chapter 3, the instance should be discarded. But the analyzer does not know if the method call is in any way related to the instance.

The second situation can be illustrated in the following example. Consider a target `a(X) b(X)` and a program trace `a(1) b(2) b(1)`. After `a(1)`, an instance is created with `X` set

to 1. When `b(2)` is called, the value of `X` conflicts with the value stored in the instance. However, the instance should not be discarded as we can see that a matching call exists later in the trace.

An example of the third situation. Consider target `a(X) b(X)` and a program trace `a(1) a(2) b(1) b(2)`. Intuitively, there should be two instances detected, one with `X` set to 1, and one with `X` set to 2. After `a(1)`, the first instance with `X=1` is created. When `a(2)` is accepted, the instance cannot be discarded even though it belongs to the alphabet of the target.

The problems described above mean that the analyzer will never discard an already running instance, the only option is to advance it. Another option is to modify the behavior in the third situation so that the analyzer will try to guess whether a method call belongs to the current instance or not. With simple contracts, the decision can be easy. Consider the target from the previous paragraph: `a(X) b(X)`. In this case, every time a method `b()` is called, the analyzer can decide whether it belongs to the currently tracked instance or not based on the value of `X`. The decision is less clear when a target contains the same method multiple times with different parameters. For example, consider target `a(X,Y) b(X) b(Y)` and a program trace containing two interleaved instances with different parameters: `a(1,2) a(2,3) b(2) b(1) b(2) b(3)`. After `a(1,2)`, a new instance is created with `X=1` and `Y=2`. After `a(2,3)`, another instance is created with `X=2` and `Y=3`. When `b(2)` is encountered, the second instance is advanced, because it matches the target. The analyzer may however discard the first instance because `b(2)` is contained in the target as `b(Y)`, but the expected method was `b(X)`. It is not clear, what the proper behavior should be. The analyzer should therefore never discard running instances.

4.2.3 Kleene Star in Contract Definition

The analyzer never invalidates a running instance. This fact allows for optimization in contract definitions. As defined in Chapter 3, a target or a spoiler is a regular expression over methods. The analyzer should therefore recognize contracts defined using all three basic operations: concatenation (`ab`), alternation (`a|b`), and Kleene star (`a*`). Due to the fact, that no method call can invalidate a running instance, the Kleene star operation is not needed. All parts of an expression that are also operands of a Kleene star operation can be removed with no impact on the analysis. For example, a regular expression `ab*c` can be replaced with `ac`. Calls to method `b` will be simply ignored. These simplified regular expressions, when converted to a finite automaton, do not create any loops. This allows for simpler structures in the implementation of the analyzer.

4.2.4 Nested Method Calls

The method described in [3] is based on program traces where every method represents a single event in the trace. However, the RoadRunner framework produces two events for every method: method entry and method exit. For parametric contracts, we need to obtain values of parameters and also the return value, which is available only on method exit. For convenience, the analyzer should use only the method exit event. This means that the analyzer may produce unexpected results when the program under analysis contains nested calls to methods that are part of the contract. Consider the following methods that are both parts of a contract:

```
public void a() { b(); }
public void b() { ... }
```

After calling method `a()`, the trace recorded by the analyzer will be `b() a()` instead of more intuitive `a() b()`.

4.3 Changes to Instrumentation Performed by RoadRunner

The RoadRunner framework does not expose the method arguments or the return value through its API. For the tracking of parametric contracts, it is necessary to obtain method arguments and return values so that the contract parameters can be assigned values.

The `enter` and `exit` of RoadRunner's `Tool` class methods both take a `MethodEvent` parameter containing the following information:

- Target — `null` for static methods, the value of `this` for instance methods.
- A `MethodInfo` object — static information about the method definition (name, descriptor, whether it is synchronized or static).
- Call site location — where was the method invoked.

The `MethodEvent` class was extended for storing method arguments and the return value. The following methods were added:

```
public Object[] getArgs();
public void setArgs(Object[] args);
public Object getReturnValue();
public void setReturnValue(Object returnValue);
```

Arguments and return values which are reference values (class instances or arrays) can be stored directly in the `Object` data type. Primitive values (such as `int` or `float`) cannot be stored in `Object` directly, they must be wrapped in a class instance. Each primitive type has a corresponding object wrapper class, for example, `int` is wrapped in the `java.lang.Integer` class. The `getArgs()` method should return an array of size 0 for a method that takes no arguments and it should return `null` when the arguments are not available. The `getReturnValue()` should return `null` when the value is not available (for example on method entry), when the method throws an exception, or when the method returns `void`.

4.3.1 Parameter Matching

When trying to advance a running instance, method arguments must be checked, if they match the previously assigned parameters. For reference types, the equality operator (`==`) is used which compares the addresses of both objects. For primitive values that are wrapped in an object, the `equals()` method must be used. As a result, the analyzer will always compare instances of wrapper classes (such as `Integer`) by calling the `equals()` method, even if the instance was created by the program under analysis. This may or may not be the intended behavior and the user of the analyzer should be aware of this.

4.4 Contracts Definition and Parsing

Before the start of the analysis, a contract must be specified. This section describes the syntax of a contract configuration file, how it is parsed, and how it is represented in the analyzer.

```
Map.put(ID)V M(K,_) Map.get(I)D M(K)
    <- Map.put(ID)V M(K,_) | Map.remove(I)D M(K) ;

Map.contains(I)Z M(K)
  ( Map.put(ID)V M(K,_) | Map.get(I)D M(K) | Map.remove(I)D M(K) )
  <- Map.put(ID)V M(K,_) | Map.remove(I)D M(K) ;
```

Listing 4.1: Contract from Chapter 3 written for a Map with `int` keys and `double` values. The first target matches an inserting element to the map at a given key and then retrieving it. The target can be invalidated by calling either `put` or `get` with the same key. The second target matches checking if a key is present in the map and then modifying the value at the given key. The target can be invalidated by replacing the value or by removing it.

4.4.1 Contract Definition Syntax

The analyzer takes a contract definition as a parameter. At the top level, the definition contains pairs of targets and spoilers. Each target and spoiler is represented by a regular expression over methods. The on-the-fly dynamic analysis described in Chapter 3 expects several spoilers to be assigned to a single target. In practice, the spoilers can be merged into a single regular expression so that each target has exactly one spoiler. Each method is parametrized, including arguments, return value, and the object it is called upon (`this`).

To specify methods unambiguously, method names must be fully qualified (for example `java/lang/Object.toString`). Java allows method overloading, so to distinguish methods with the same name but different number and type of their parameters, the contract definition contains the method descriptor (for example `(Ljava/lang/Object;)V`. Each method in a contract definition consists of the enclosing class, the method name, the method descriptor, and a list of meta-variables. For example:

```
test/sanity/ArrayList.set(ILjava/lang/Object;)Ljava/lang/Object; X=Y(Z,_)
```

This represents the `set` method called on an `ArrayList` from the `test.sanity` package. The first meta-variable `X` is the return value, `Y` is the `ArrayList` instance, and `Z` is the first parameter (integer). The second parameter (object) is marked with a free meta-variable (`_`) meaning that the analyzer will ignore its value. The return value or the target of the method may be omitted, the parser will treat them as free meta-variables.

Targets and spoilers are defined using a limited regular expression over methods. Concatenation is achieved simply by writing two methods, one after another, alternation is denoted by a vertical bar (`|`). The Kleene star operator (`*`) is not allowed (see Section 4.2.3). An example of a target definition:

```
Test.a()V X() ( Test.b()V X() | Test.c()V X() )
```

A contract clause is defined as two regular expressions over methods, separated by an arrow and a semicolon at the end. A contract is then a list of clauses. An example of a full contract is shown in Listing 4.1. The full grammar for contracts is shown in Appendix C.

4.4.2 Contract Representation

All parts of a contract definition are parsed into a corresponding class instance. Each method in a target or spoiler is represented by a `Signature` instance. It consists of a method name, a fully qualified name of the enclosing class, and a method descriptor. The method

descriptor contains information about parameter types. The main purpose of this class is to be compared with method invocation events to determine if the invoked method matches the method from a contract. Meta-variables are parsed into a `MetaVars` instance.

Each target and spoiler is represented by a `State` instance that will later be used to construct a finite automaton for detecting running target and spoiler instances. A target with a single method will be represented by two states: one starting state with a transition to an accepting state. The transition will contain the method signature and meta-variables. During parsing of more complex targets and spoilers, these one-method state structures are combined. See Figure 4.1 for an example. During the analysis, `State` instances will be used for checking whether a given method invocation can advance a given target or spoiler.

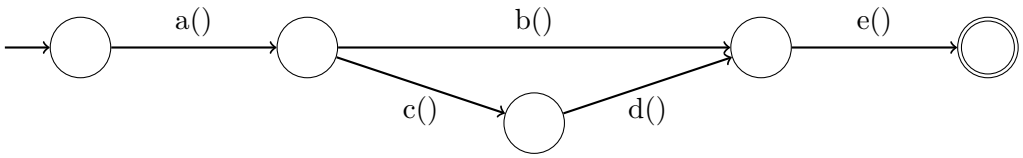


Figure 4.1: A structure of states representing the following regular expression: `a(b|cd)e`.

A contract is then made up of target–spoiler pairs. When a contract is created, a set of all method signatures used in the contract is extracted. During the analysis, the set of signatures is used to filter method invocations so that only relevant methods are processed by the analyzer.

4.5 Contract Analyzer

The main class, `ContractAnalyzer` receives events from the program under analysis, detects target and spoiler instances in all threads, and looks for contract violations when a target or spoiler instance finishes. The `ContractAnalyzer` class provides the following interface:

- A constructor that takes a `Contract` instance.
- `exit` method that is called on method exit. It takes a thread identifier, a method signature, and method arguments (also containing the return value). When a contract violation is detected, the `exit` method will throw an exception.
- `acquire` and `release` methods that are called when a synchronized block or a synchronized method is entered and exited. The methods both take a thread identifier and a lock identifier.
- `create` method that is called when a new thread is created. It takes a thread identifier.
- `fork` and `join` methods that take two thread identifiers.

The `ContractAnalyzer` class manages data stored with threads and locks. When created by calling `create`, each thread will get a trace window and a vector clock. Methods `acquire`, `release`, `fork`, and `join` only modify vector clocks of threads and locks.

The `exit` method calls the trace window associated with a given thread. The trace window receives method signature and arguments so that it can try to advance all of its

target and spoiler instances. It also receives the current vector clock and references to trace windows of other threads so that it can look for contract violations.

The `ContractAnalyzer` class is created and called by the `ContractTool` class which directly uses the RoadRunner API. The `ContractTool` class is described in Section 4.6. The interface of the analyzer is made up of synchronized methods to ensure proper synchronization. This approach is not the same as the event serialization mentioned in Chapter 2. All events that are not calling analyzer's methods will still be concurrent.

4.5.1 Tracking of Target and Spoiler Instances

For each thread, a `Window` object is kept during the analysis. It contains target and spoiler instances present in a trace window. When a method invocation is detected in a thread, all target and spoiler instances are advanced (if possible) and new instances are started.

An instance is bound to a target or spoiler from the contract definition. The instance is created by encountering the first method signature in a target or spoiler. The instance is advanced to the next state and waits for the next method as defined in the given target or spoiler. During the first transition, values of all parameters are assigned. There can be multiple instances of the same target or spoiler that vary only by the value of their parameters. Each instance remembers the vector clock of its beginning.

When an instance is advanced using the last method in the target or spoiler definition, it reaches an accepting state. At this point, the analyzer checks for contract violations (see Section 4.5.2). Then the instance is reset. That means that the instance again waits for the first method signature in a given target or spoiler. The value of parameters will however stay unchanged. The vector clocks of the beginning and the end of the instance are saved. So when an instance is running for the second time, it has access to the vector clocks of a previously encountered instance. See Figure 4.2 for illustration of an instance life cycle.

When a method call enters the trace window, all running instances (those with parameters already assigned) are advanced. The method call may however also start a new instance that is not yet part of the trace window. The analyzer tries to create new instances from targets and spoilers from the contract. These new instances are added to the trace window only if there is no matching instance already present in the trace window. Two instances are matching, if they are bound to the same target or spoiler, their parameters share the same values, and they both just started (they accepted the same first method). In practice, the only difference between these matching instances is that one contains information about the previously encountered instance while the other does not.

Example. Consider target `a(X) b(X)` and a program trace `a(1) b(1) a(2) b(2) a(1)`. After `a(1)`, the analyzer adds a new instance i_1 to the window with $X = 1$. After `b(1)`, i_1 is advanced, accepted, and reset. No new instance is started because there is no target starting with method `b`. After `a(2)`, i_1 cannot be advanced, because the parameters do not match. New instance i_2 with $X = 2$ is added to the trace window. After `b(2)`, i_2 is advanced, accepted, and reset. When `a(1)` is encountered again, i_1 is advanced but no new instance is added because i_1 would match the newly created instance.

4.5.2 Detection of Contract Violations

Each time a target or a spoiler is fully recognized by the analyzer, it must be checked whether there are any contract violations. When a method enters a trace window, there may be several instances from the same thread that will be fully accepted by this method

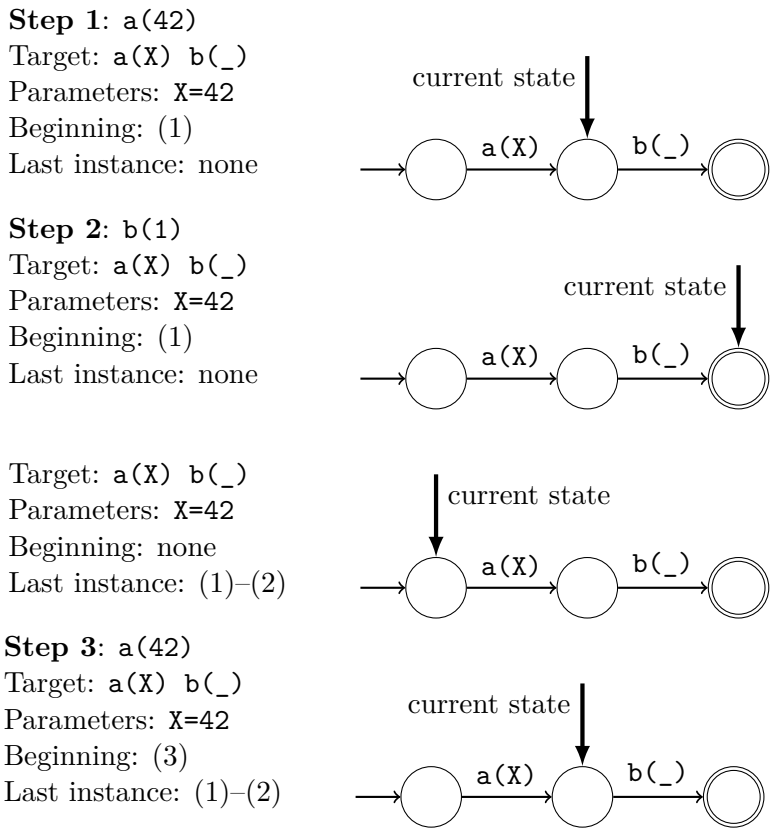


Figure 4.2: An example of an instance life cycle. In the first step, the instance is created when a method call `a(42)` is encountered. The vector clock of the beginning is set and the parameters are assigned a value. In step 2, the instance is fully accepted. At this point, the analyzer looks for contract violations. Then the instance is reset, the vector clock of the beginning is reset, and the vector clocks of the last instance are set. In step 3, method `a` is called with an argument that matches the value of `X` in the instance. The instance is started again.

call. For each accepted target and spoiler instance, the analyzer will look for conflicting instances in trace windows from other threads.

When an instance is fully accepted, the analyzer will retrieve all *conflicting instances* from other threads. If the instance is a target instance, the analyzer will retrieve instances of a spoiler that can invalidate the target, as specified in the contract. Similarly, if the instance is a spoiler instance, the analyzer will look for instances of a target that can be invalidated by the spoiler. For each conflicting instance, it is checked whether the contract parameter values match. Then finally, the analyzer checks the vector clocks of each matching instance and decides if the two instances interleave each other.

The two instances may not actually interleave each other. The interleaving is decided based on the vector clocks of the beginnings and ends of the two instances. Vector clocks are only updated when threads synchronize themselves. So the analyzer will mark the two instances as interleaving when there was not a synchronization between the threads that

would prevent them to interleave each other in another run of the program under analysis. Figure 4.3 shows an example of a contract violation detected in a program.

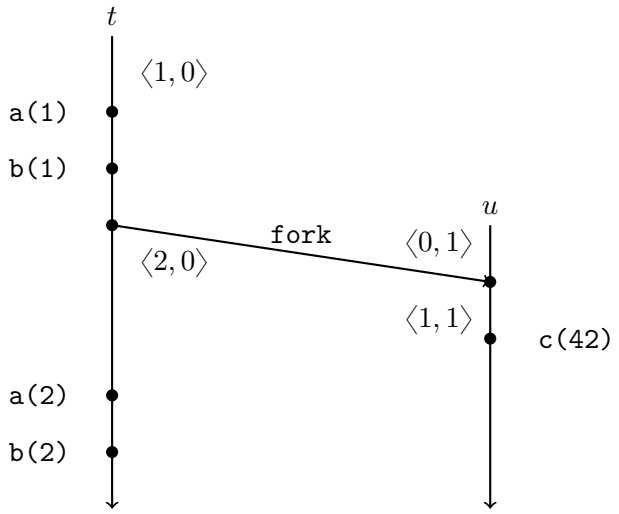


Figure 4.3: An example of a contract violation in a program. Consider a target `a(X) b(X)` and a spoiler `c(_)`. In thread t, a target instance r_1 with $X = 1$ is detected, it begins and ends at $\langle 1, 0 \rangle$. There is no spoiler instance accepted at this moment, so there is no contract violation. Then a thread u is created and a spoiler instance s_1 is accepted, it begins and ends at $\langle 1, 1 \rangle$. The spoiler instance does not interleave the r_1 because $\langle 1, 0 \rangle \sqsubseteq \langle 1, 1 \rangle$. Then another target instance r_2 is detected in t with $X = 2$, it begins and ends at $\langle 2, 0 \rangle$. The instance interleaves with spoiler instance s_1 because $\langle 1, 1 \rangle \not\sqsubseteq \langle 2, 0 \rangle$.

4.6 A Contract Analyzer Tool

The `ContractAnalyzer` class described in Section 4.5 is not meant to be called directly by RoadRunner because it contains only methods specific to the validation of contracts. The `ContractTool` class was created as a layer between RoadRunner and `ContractAnalyzer`. Its purpose is to load a contract from a file, create a `ContractAnalyzer` instance, and forward relevant events from RoadRunner to `ContractAnalyzer`.

The `ContractTool` class is a subclass of the `Tool` class (see Section 2.6). The following methods are overridden:

- `init`, which is called before the analysis, after command-line options are processed;
- `exit`, which is called when a method exits;
- `create`, which is called when a new thread is created;
- `acquire` and `release()`, which is called when a synchronized method or a synchronized block is entered or exited;
- `preStart`, which is called before a new thread is started (after a fork operation);
- `postJoin`, which is called after a thread is joined with another thread.

Methods for detecting memory accesses will not be used by the tool, and also no data will be stored in shadow memory locations, except for vector clocks stored with locks. In its `init` method, the tool loads a contract definition from a file and calls the contract parser. After the contract is parsed, it constructs `ContractAnalyzer`.

The `exit` method extracts relevant data from `MethodEvent` that is supplied by RoadRunner. Then it checks if the method exists in the contract, and if it does, the method call is forwarded to `ContractAnalyzer`. The method filtering happens in `ContractTool` and not in `ContractAnalyzer` because it is not essential to the analysis and also, there are several filtering solutions available. In this Master's thesis, the filtering is done by comparing method signatures during the analysis. A better approach, which might be implemented in the future, is to instrument only methods contained in a contract. In that case, no filtering during analysis is needed as it is guaranteed that each method event produced will belong to a method in the contract. The rest of the methods, `create`, `acquire`, `release`, `preStart`, and `postJoin`, just pick relevant information from events provided by RoadRunner and forward it to `ContractAnalyzer`.

Chapter 5

Implementation and Testing

The analyzer described in Chapter 4 was successfully implemented. This chapter provides implementation details and clarifies decisions made during the implementation. Section 5.1 describes approaches and principles that shaped the implementation. Before the implementation took place, several RoadRunner dependencies were upgraded, see Section 5.2. In Section 5.3, the contract file parser is described. Changes to the instrumentation performed by RoadRunner are described in detail in Section 5.4. Finally, Section 5.5 describes testing approaches.

5.1 General Approaches

The implementation of the analyzer was guided by several principles or approaches that are described in this section.

5.1.1 Functional Programming

The analyzer implementation uses several concepts from functional programming which are briefly described in this section.

Immutable data structures All classes are immutable, except for `ContractAnalyzer` and `ContractTool`. Once created, the internal state of objects does not change. All operations produce a new object instead of modifying the current one. See the next section for more details.

Side effects and pure functions Most methods in the analyzer are pure functions, which means that each call can be always replaced with a resulting value of the call. For example, consider pure function `int sum(int a, int b)`. Then the method call `sum(2,3)` can be always replaced with `5` without changing program behavior. Pure functions perform no side effects (such as writing to a file).

Higher-order function The analyzer contains and uses methods that take functions as parameters.

5.1.2 Immutable Data Structures

The data structures used by the analyzer are *immutable*. An immutable object is an object whose internal state remains constant after it has been created. This property brings several benefits. The object can be shared among multiple threads without the need for synchronization. Immutable objects are always in a consistent state. The public methods of a given class will always behave the same way on an object. Immutable objects are also easy to test and reason about.

In Java, immutability is achieved by following rules. Immutable classes should be `final` to avoid overriding of methods. All fields of a class should be `private` and `final`. All fields should reference only immutable objects or, if not possible, the fields should not be modified in the class. All methods that would normally modify the object should return updated object instead. See Listing 5.1 for an example. To use immutable objects effectively, the Vavr collection library[1] was used. It provides immutable replacements for standard Java collections.

```
public final class FiniteAutomaton {
    private final State start;
    private final State current;

    public FiniteAutomaton(State start) { this(start, start); }

    private FiniteAutomaton(State start, State current) {
        this.start = start;
        this.current = current;
    }

    public FiniteAutomaton advance(Signature sig, Args args) {
        return new FiniteAutomaton(start, current.advance(sig, args));
    }

    public FiniteAutomaton reset() {
        return new FiniteAutomaton(start, start);
    }

    public boolean isRunning() { return current != start; }
}
```

Listing 5.1: Simplified implementation of an immutable finite automaton. The automaton consists of references to the starting state and the current state. The public constructor allows creating automatons that are in their starting states, ensuring consistency. The advance method does not update the current state but creates a new finite automaton with an updated current state.

5.1.3 Dependency Inversion Principle

Several parts of the analyzer were designed with the dependency inversion principle in mind. Classes holding low-level data, such as method signatures, method arguments, or contract

[1] available at https://www.vavr.io/

parameters, are not used directly by the analyzer but via interfaces. For example, there is the `JvmSignature` class that implements the `Signature` interface.

The collection that stores target and spoiler instances in a trace window is abstracted in the `InstanceCollection` interface. The `MultimapInstanceCollection` implements the collection using Vavr's `Multimap` data structure. This approach makes it easy to create alternative implementations of the collection.

5.2 ASM 7.0 and Java 11

The analyzer was built on RoadRunner version 0.5 from 2017. It contains the following dependencies: the ASM framework in version 5.0.2 with custom modifications, JFlex in version 1.4.2, and CUP in version 11a. The project was written for Java 8 and was built using Ant. For an easier implementation of the analyzer, several components were upgraded.

The ASM framework was upgraded to version 7.0 which supports Java 11. RoadRunner can therefore analyze programs compiled for Java 11. Before the upgrade took place, the custom modifications of the ASM framework were isolated to a series of patches against the unmodified ASM version 5.0.2. Then, for each new version up to 7.0, the ASM was always replaced with a newer version and the custom patches were reapplied and modified if necessary. As a result, the ASM framework can be easily upgraded in the future by reapplying the custom patch series. The RoadRunner itself was modified to be built for Java 11.

5.3 Contract File Parsing

One of the inputs to the analyzer is a contract definition. The syntax of the definition is described in Chapter 4. Due to its length, it is passed to the analyzer in a text file. The file name is specified using a command-line option. RoadRunner allows tools to easily add new command-line options. The options are then automatically parsed and made available for the tool to use.

The file with contract definition is then scanned using a lexical analyzer and parsed using a LALR parser. The lexical analyzer is generated using JFlex[2] and the parser is generated by CUP[3]. The primary reason for choosing these generators was that both of them were already used in RoadRunner, so no new dependency was added to the project.

The lexical analyzer recognizes various symbols for delimiting the methods in a contract but it does not split class names and method descriptors, they are passed to the parser as a single string. For example, `java/lang/Object` or `(Ljava/langString;II)V`. The list of terminals it produces is defined in the parser.

The parser takes the contract definition file contents, creates a lexical analyzer, and parses the file. The result is a `Contract` instance or an exception. Each method in a contract is parsed as a finite-state machine with a single state. The whole target or spoiler definition is constructed either by concatenating or alternating two states from left to right.

The current implementation of the parser introduces a limitation to the range of allowed regular expressions. When the alternation operation is used (denoted by `|`), the expressions cannot start with the same method. For example, the regular expression `(ab|ac)` is not allowed. The resulting finite automaton would be nondeterministic. The current imple-

[2] available at https://jflex.de/
[3] available at http://www2.cs.tum.edu/projects/cup/

mentation does not perform any conversion, all expressions must be converted by the user. In the previous example, the expression would need to be converted to `a(b|c)`.

5.4 Changes in Instrumentation

RoadRunner was modified to obtain method arguments and return values during the analysis. The implementation is based on changes described in [9], the final version however fixes several major issues. The initial implementation adds new fields to the `MethodEvent` class which holds information about a method invocation. These changes were taken without modifications from [9].

The main instrumentation logic is contained in the `SyncAndMethodThunkInserter` class from the `rr.instrument.classes` package, in the `createMethodThunk` method. The method creates a new method that will generate enter and exit events and call the original method. In the beginning, the values of method parameters need to be stored, at the end, the return value must be stored. The initial implementation described in [9] contained several issues. Static methods could not be instrumented because of incorrect indexing of local variables. Methods with parameters of type `double` or `long` could not be instrumented, because the implementation was not taking into account that these values occupy two local variables. These issues have been fixed and an extensive test suite was created to verify the final implementation.

Each method is instrumented as follows. A new array of type `Object` is allocated with the size equal to the number of parameters (taken from the method descriptor). Then for each parameter, its value is loaded onto the operand stack. Reference values are stored directly in the array. Primitive values are wrapped in an object by calling the `valueOf()` method in the corresponding class depending on the primitive type. For example, `int` values are passed to the `java.lang.Integer.valueOf()` function. After processing all arguments, the array is stored in a local variable.

The original method is then called in a try–catch block. On normal exit, the return value is converted to an object, the same way parameters are converted. Then a method exit event is generated, containing both the array of arguments and the return value. If an exception is caught, the return value is set to `null` and an exit event is generated containing the arguments. An example of a instrumented method is shown in Listing 5.2.

5.5 Testing

Each part of the analyzer was thoroughly tested using several different approaches. The core analyzer functionality was tested using unit tests written in JUnit 5[4]. The instrumentation of method arguments and returns values was tested using a custom RoadRunner tool. The integration of all parts was tested using Bash scripts that prepare contract files, programs under analysis, and launch RoadRunner. This section describes approaches used for testing different parts of the analyzer. See Appendix B for instructions for running the tests.

The analyzer was implemented so that there is almost no need to work with RoadRunner's internal structures when testing the analyzer. The `ContractTool` class serves as a wrapper for `ContractAnalyzer`. All structures, such as `MethodEvent`, are transformed into objects specific to the contract analyzer. In tests, `ContractAnalyzer` is used directly.

[4]available at https://junit.org/junit5/

```
public int foo(int, java.lang.String);
 0: invokestatic  #20   // Method rr/state/ShadowThread
                        // .getCurrentShadowThread:()Lrr/state/ShadowThread;
 3: astore        5
 5: aload_0
 6: sipush        508
 9: aload         5
11: invokestatic  #27   // Method rr/tool/RREventGenerator
                        // .enter:(Ljava/lang/Object;ILrr/state/ShadowThread;)V
14: iconst_2
15: anewarray     #3    // class java/lang/Object
18: astore_3
19: aload_3
20: iconst_0
21: iload_1
22: invokestatic  #33   // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
25: aastore
26: aload_3
27: iconst_1
28: aload_2
29: aastore
30: aload_0
31: iload_1
32: aload_2
33: invokespecial #35   // Method __$rr_foo__$rr__Original_:(ILjava/lang/String;)I
36: dup
37: invokestatic  #33   // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
40: astore        4
42: aload         5
44: aload_3
45: aload         4
47: invokestatic  #39   // Method rr/tool/RREventGenerator.exit:
                // (Lrr/state/ShadowThread;[Ljava/lang/Object;Ljava/lang/Object;)V
50: goto          61
53: aload         5
55: aload_3
56: aconst_null
57: invokestatic  #39   // Method rr/tool/RREventGenerator.exit:
                // (Lrr/state/ShadowThread;[Ljava/lang/Object;Ljava/lang/Object;)V
60: athrow
61: ireturn
```

Listing 5.2: A method instrumented to obtain arguments and the return value. On lines 5–11, the enter event is generated. On lines 14–18, an array for arguments is created and stored in a local variable. On lines 19–25, the first argument is wrapped in an object and stored in the array on index 0. On lines 26–29, the second argument is stored directly in the array on index 1. On lines 30–33, the original method is called. On lines 36–40, the return value is wrapped and stored in a local variable. Lines 42–50 generate the exit event. Lines 53–60 contain a catch block in case an exception is thrown in the original function.

RoadRunner is not executed, the events coming from the program under analysis are created by tests, allowing for tests that do not rely on the thread scheduler and are very fast.

The contract parser is tested using JUnit 5 tests. The contract definition is passed to the parser and the produced `Contract` instance is compared with a `Contract` instance constructed directly in code. The contract for comparison is constructed by creating appropriate objects such as method signatures, meta-variables, and finite automaton states.

Changes to instrumentation were tested by preparing a custom subclass of RoadRunner's `Tool` class that overrides the `exit` method and prints arguments and the return value to the standard output. A testing Java program was created that calls methods with various numbers and types of parameters. The test consists of analyzing the testing program with RoadRunner using the custom tool. The tool prints method arguments and return values to the output and the values must match those in the testing program. The whole process is automated using a Bash script.

The integration of all parts is again automated using Bash scripts. Testing programs and files with contract definitions are prepared. The script compiles the testing program and analyzes it with RoadRunner that is using the contract tool. Then it verifies if a contract violation has been found.

5.5.1 Overview of Integration Tests

This section provides an overview of integration tests written for the analyzer.

Array list The contract in this test covers operations on an array list with the following operations: `add`, `get`, `set`, `contains`, `indexOf`, `remove`, and `size`. There are four programs in this test, each violating one contract clause.

Account This test was taken from the test suite of the Gluon project[5]. The test simulates a bank account with two operations: `getBalance` and `setBalance`. Even though the operations are synchronized, there is a high-level data race where a thread reads the balance, increments it, and writes it back.

Block allocation This test was taken from the test suite of the Gluon project. There is a shared vector that for each block of a buffer stores whether it is free or occupied. When allocating a block, a free block must be found and then it must be set as occupied. Between finding a block and marking it as occupied, another thread may mark the same block as occupied.

Arithmetic database This test was taken from the test suite of the Gluon project. The test simulates a database with two tables. The first table contains a set of regular expressions and the second holds the results of the expressions. Each table is accessed using synchronized methods. There are several problems related to the absence of transactions when accessing multiple tables or when performing several operations on the same table.

Connection test This test was taken from the test suite of the Gluon project. The test simulates a chat application that uses a socket to send messages over the network.

[5] available at https://github.com/trxsys/gluon

A message counter is associated with the socket that is incremented with each message sent. When the socket is closed, there is an inconsistency when a thread may see a closed socket but the counter is not yet zeroed. When a message is sent, it is checked that the socket is still open before sending the message. However, the socket may be closed in the meantime by another thread.

5.6 Performance

The performance of the analyzer was checked on the Account test case (see Section 5.5) with several modifications. The high-level data race present in the test case was removed so that the analyzer will not stop the program right after a contract violation is detected. The number of operations performed on the account was added as a parameter. The Account test case starts two threads and all tracked operations are performed on a single object, so the number of tracked target and spoiler instances is constant.

The test case was run using the RoadRunner benchmark mode. The measured results do not include the initialization of the analyzer. The Account test case was run three times: without instrumentation, with instrumentation, and with instrumentation and contract validation. The results are shown in Figure 5.1. As expected [6], the instrumented program is approximately ten times slower than the original program. The analyzer is 10–100 times slower than the instrumented program. To fully assess the performance of the analyzer, more benchmarks would be needed.

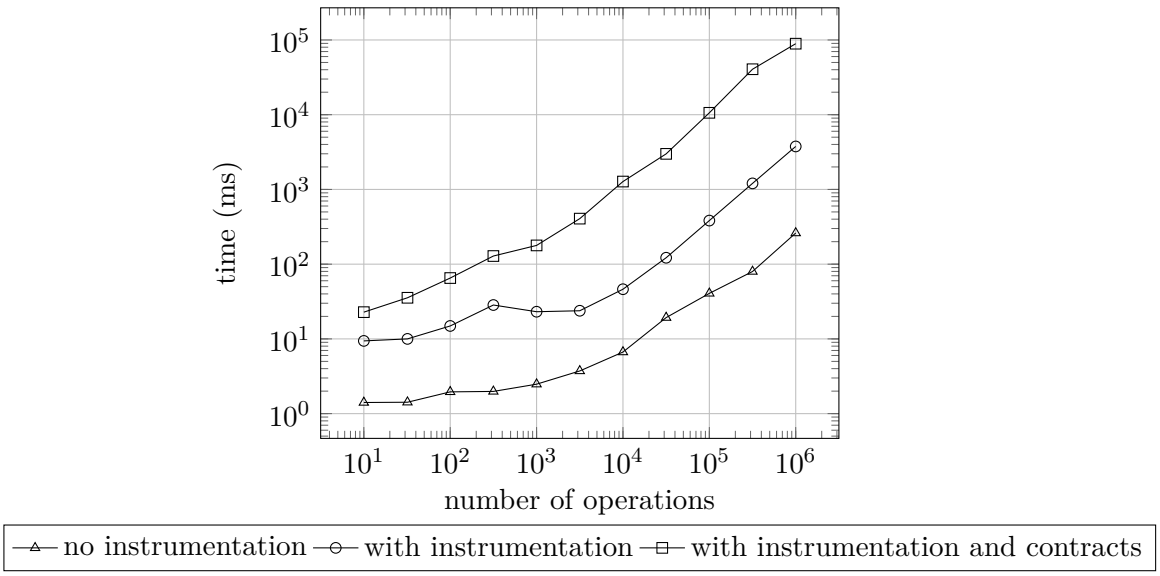


Figure 5.1: Results of the Account benchmark. The first series shows the runtime without any instrumentation. The second series is run with RoadRunner instrumentation but without tracking of targets and spoilers. The last one shows the runtime of instrumented program with contract validation.

Chapter 6

Conclusion

The goal of this thesis was to design a dynamic analyzer for validating parametric contracts with spoilers. The analyzer was fully implemented as an extension to the RoadRunner framework.

The first part of this thesis provided the necessary background in multi-threaded programming in Java, dynamic analysis, and instrumentation in the RoadRunner framework. Contracts for parallelism were then introduced together with an on-the-fly method for contract analysis. A dynamic analyzer for tracking parametric contracts was proposed. Several restrictions were put on the analyzer in the design phase to mitigate problems in previous prototype implementations. The analyzer consists of the following parts: a parser for contract definitions, modified instrumentation of methods, and the core analyzer that tracks target and spoiler instances and detects contract violations. All parts of the analyzer were implemented and their functionality was verified by an extensive test suite. The analyzer was able to detect all contract violations present in the testing programs.

The analyzer implementation provides a solid basis for contract validation of programs written in Java. There is an ongoing work on the formalization of parametric contracts and extending experiments on standard libraries. The analyzer can be used for those experiments. The changes in method instrumentation are not tied to the contract validation and can be used by various other analyzers that may benefit from obtaining method arguments and return values.

In the future, various parts of the analyzer may be tuned for better performance. The functional implementation allows easy parallelization of checking contract violations. The instrumentation can be further reduced to obtain only program actions relevant to the contract validation. Instance invalidation can be introduced by clarifying the conditions in the context of parametric contracts. The analyzer can be also combined with noise injection techniques for detecting more contract violations.

Bibliography

[1] Artho, C., Havelund, K. and Biere, A. High-level data races. *Software Testing, Verification and Reliability*. 2003, vol. 13, no. 4, p. 207–227. DOI: 10.1002/stvr.281.

[2] Bruneton, E. *ASM 4.0 A Java bytecode engineering library* [online]. 2011 [cit. 2021-01-09]. Available at: https://asm.ow2.io/asm4-guide.pdf.

[3] Dias, J. R., Ferreira, C., Fiedor, J., Lourenco, J., Smrčka, A. et al. Verifying Concurrent Programs Using Contracts. In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Institute of Electrical and Electronics Engineers, 2017, p. 196–206. DOI: 10.1109/ICST.2017.25. ISBN 978-1-5090-6032-0.

[4] Fiedor, J., Letko, Z., Lourenco, J. and Vojnar, T. Dynamic Validation of Contracts in Concurrent Code. In: *Proceedings of the 15th International Conference on Computer Aided Systems Theory*. The Universidad de Las Palmas de Gran Canaria, 2015, p. 177–178. ISBN 978-84-606-5438-4.

[5] Flanagan, C. and Freund, S. N. FastTrack: Efficient and Precise Dynamic Race Detection. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. november 2010, vol. 53, no. 11, p. 93–101. DOI: 10.1145/1839676.1839699. ISSN 0001-0782.

[6] Flanagan, C. and Freund, S. N. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In: *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. New York, NY, USA: Association for Computing Machinery, 2010, p. 1–8. PASTE '10. DOI: 10.1145/1806672.1806674. ISBN 9781450300827.

[7] Ghezzi, C., Jazayeri, M. and Mandrioli, D. *Fundamentals of Sofware Engineering*. 2nd ed. Prentice Hall, 2003. 269-336 p. ISBN 0-13-099183-X.

[8] Gosling, J., Joy, B., Steele, G. L., Bracha, G. and Buckley, A. *The Java Language Specification, Java SE 8 Edition*. 1st ed. Addison-Wesley Professional, 2014. 645-659 p. ISBN 013390069X.

[9] Janoušek, M. *Dynamic Analyzers for SearchBestie Platform*. Brno, CZ, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology.

[10] Letko, Z. *Analysis and Testing of Concurrent Programs*. Brno, CZ, 2012. Ph.D. thesis. Brno University of Technology, Faculty of Information Technology.

[11] Lindholm, T., Yellin, F., Bracha, G. and Buckley, A. *The Java Virtual Machine Specification, Java SE 8 Edition*. 1st ed. Addison-Wesley Professional, 2014. 69-332 p. ISBN 013390590X.

[12] Schildt, H. *Java: The Complete Reference, Eleventh Edition*. 11th ed. New York: McGraw-Hill Education, 2018. ISBN 1260440230.

[13] Sousa, D. G., Dias, R. J., Ferreira, C. and Lourenço, J. Preventing Atomicity Violations with Contracts. *CoRR*. 2015, abs/1505.02951.

Appendix A

Storage Medium

The contents of the enclosed CD:

`RoadRunner/` The source code of the analyzer.

`RoadRunner-compiled/` Compiled analyzer.

`patches/` The source code of the analyzer, as patches against commit `b141616` in the upstream RoadRunner repository[1].

`contracts-for-concurrency.pdf` The text of the thesis.

`contracts-for-concurrency-print.pdf` The text of the thesis, for color printing.

`contracts-for-concurrency/` The sources code of the text of the thesis.

[1] available at https://github.com/stephenfreund/RoadRunner

Appendix B

Manual

The analyzer requires a Java Virtual Machine version 11 and Ant[1]. For convenience, the analyzer should be compiled and run on Linux as there are several Bash scripts used in the process. All commands, such as `ant` or `javac`, that are run during the compilation should belong to the same JVM installation.

On Fedora 34, install Java 11 JDK and Ant by running:

```
$ sudo dnf install java-11-openjdk ant
```

On Ubuntu 21.04, run:

```
$ sudo apt install openjdk-11-jdk
```

Ant must be installed locally from the project's website. To compile the project, run `ant` in the RoadRunner directory. The project should compile and print `BUILD SUCCESSFUL` at the end. The unit tests can be run at this step with `ant test`. The tests should all pass.

Before running RoadRunner, edit the `msetup` file. On line 36, edit the path to the JVM installation. On Fedora 34, `/usr/lib/jvm/java-11` should be used. On Ubuntu 21.04, `/usr/lib/jvm/java-11-openjdk-amd64` should be used. Then run `source msetup`. The environment variables should be properly exported.

To verify the compilation, run `rrrun -help`. To instrument and run a testing program, run the following commands:

```
$ javac test/Test.java
$ rrrun test.Test
```

To launch the contract analyzer on a simple program, run the following commands:

```
$ javac test/ContractTest.java
$ rrrun -tool=CT -contractFile=test/ContractTest.contract test.ContractTest
```

The analyzer should find a contract violation. The integration tests are run by the following command:

```
$ testScripts/all.sh
```

[1] available at http://ant.apache.org/

Appendix C

Contract Definition Grammar

The configuration files with contract definitions must follow the grammar presented below. The grammar is described in the BNF syntax.

```
<contract> ::= <clause> | <contract> <clause>

<clause> ::= <method_expr> "<-" <method_expr> ";"

<method_expr> ::= <method>
                | "(" <method_expr> ")"
                | <method_expr> "|" <method_expr>
                | <method_expr> "(" <method_expr> ")"
                | <method_expr> <method>

<method> ::= <class> <name> <descriptor> <metavars>

<metavars> ::= <metavar> "=" <metavar> "(" <metavars_list> ")"
    | <metavar> "=" <metavar> "()"
    | <metavar> "=("  <metavars_list> ")"
    | <metavar> "(" <metavars_list> ")"
    | "(" <metavars_list> ")"
    | <metavar> "()"
    | "()"

<metavars_list> ::= <metavar> | <metavars_list> "," <metavar>

<metavar> ::= <letter> | "_"
```

The terminals have the following definitions:

- `<class>` is a class name in the internal representation of the JVM specification, for example: `java/lang/Object`;
- `<name>` is the method name, as defined by the JVM specification, for example: `equals`;
- `<descriptor>` is the method descriptor, as defined by JVM specification, for example: `(Ljava/lang/Object;)Z`;
- `<letter>` is a single letter, such as `X`.

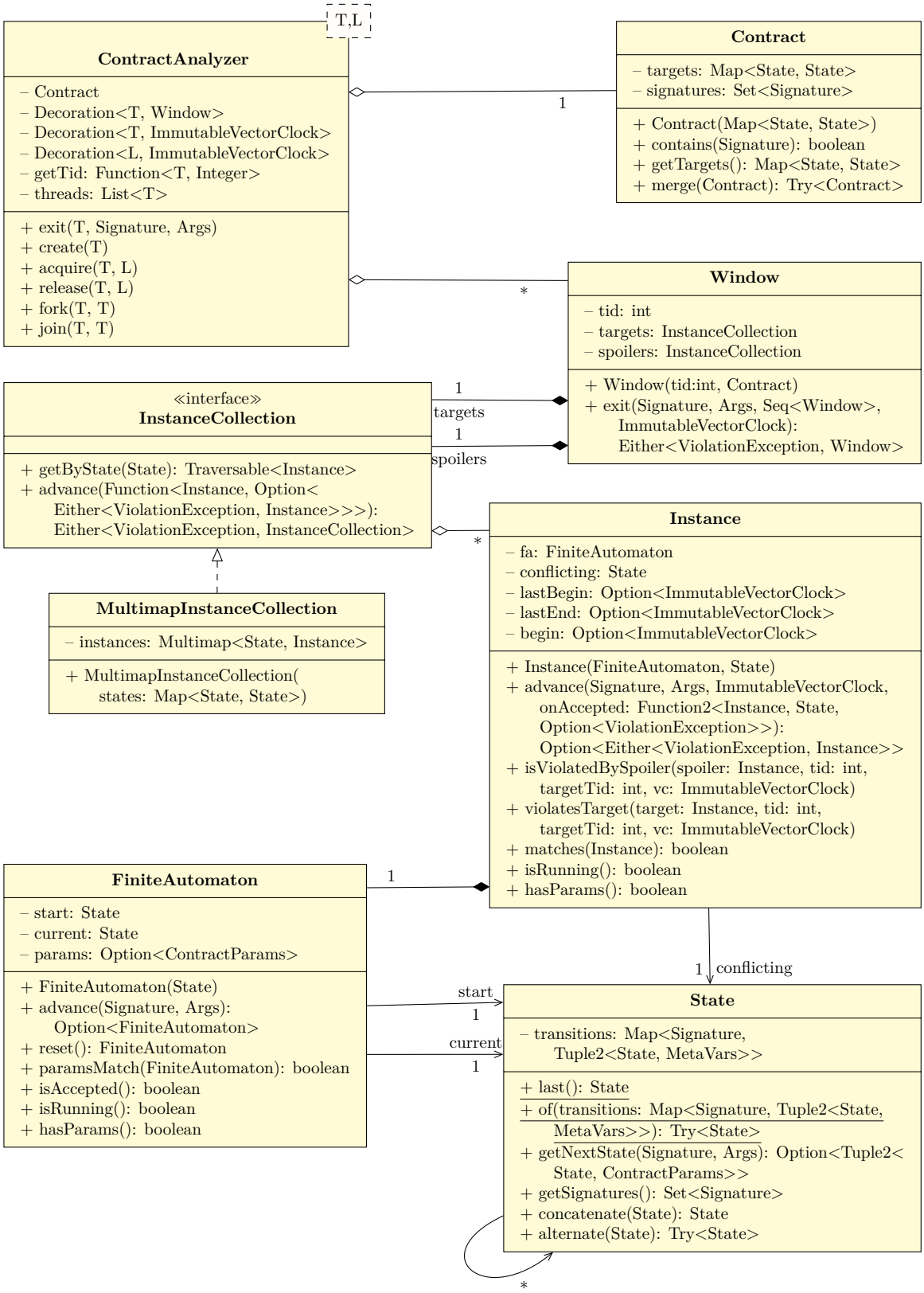
Appendix D

Class Diagram of the Contract Analyzer

The following page contains a UML diagram of the contract analyzer described in Chapter 3. In the diagram, the following classes are omitted for clarity:

- the `ContractTool` class which connects `ContractAnalyzer` to RoadRunner,
- the contract lexer and parser,
- basic classes and interfaces used for holding simple data, such as `Args`, `Signature`, `MetaVars`, `ImmutableVectorClock`, and `ContractParams`.

The `ContractAnalyzer` class is a generic class with two type parameters: `T` is a type representing a thread, `L` represents a lock. During the analysis, RoadRunner types are used: `T` is `ShadowThread`, `L` is `ShadowLock`. Collections and containers used in the diagram are parts of the Vavr library.


T,L
ContractAnalyzer
– Contract
– Decoration<T, Window>
– Decoration<T, ImmutableVectorClock>
– Decoration<L, ImmutableVectorClock>
– getTid: Function<T, Integer>
– threads: List<T>
+ exit(T, Signature, Args)
+ create(T)
+ acquire(T, L)
+ release(T, L)
+ fork(T, T)
+ join(T, T)
Contract
– targets: Map<State, State>
– signatures: Set<Signature>
+ Contract(Map<State, State>)
+ contains(Signature): boolean
+ getTargets(): Map<State, State>
+ merge(Contract): Try<Contract>
1
*
Window
– tid: int
– targets: InstanceCollection
– spoilers: InstanceCollection
+ Window(tid:int, Contract)
+ exit(Signature, Args, Seq<Window>,
ImmutableVectorClock):
Either<ViolationException, Window>
«interface»
InstanceCollection
+ getByState(State): Traversable<Instance>
+ advance(Function<Instance, Option<
Either<ViolationException, Instance>>>):
Either<ViolationException, InstanceCollection>
1
targets
1
spoilers
*
Instance
– fa: FiniteAutomaton
– conflicting: State
– lastBegin: Option<ImmutableVectorClock>
– lastEnd: Option<ImmutableVectorClock>
– begin: Option<ImmutableVectorClock>
+ Instance(FiniteAutomaton, State)
+ advance(Signature, Args, ImmutableVectorClock,
onAccepted: Function2<Instance, State,
Option<ViolationException>>):
Option<Either<ViolationException, Instance>>
+ isViolatedBySpoiler(spoiler: Instance, tid: int,
targetTid: int, vc: ImmutableVectorClock)
+ violatesTarget(target: Instance, tid: int,
targetTid: int, vc: ImmutableVectorClock)
+ matches(Instance): boolean
+ isRunning(): boolean
+ hasParams(): boolean
MultimapInstanceCollection
– instances: Multimap<State, Instance>
+ MultimapInstanceCollection(
states: Map<State, State>)
FiniteAutomaton
– start: State
– current: State
– params: Option<ContractParams>
+ FiniteAutomaton(State)
+ advance(Signature, Args):
Option<FiniteAutomaton>
+ reset(): FiniteAutomaton
+ paramsMatch(FiniteAutomaton): boolean
+ isAccepted(): boolean
+ isRunning(): boolean
+ hasParams(): boolean
1
1
conflicting
start
1
current
1
State
– transitions: Map<Signature,
Tuple2<State, MetaVars>>
+ last(): State
+ of(transitions: Map<Signature, Tuple2<State,
MetaVars>>): Try<State>
+ getNextState(Signature, Args): Option<Tuple2<
State, ContractParams>>
+ getSignatures(): Set<Signature>
+ concatenate(State): State
+ alternate(State): Try<State>
*